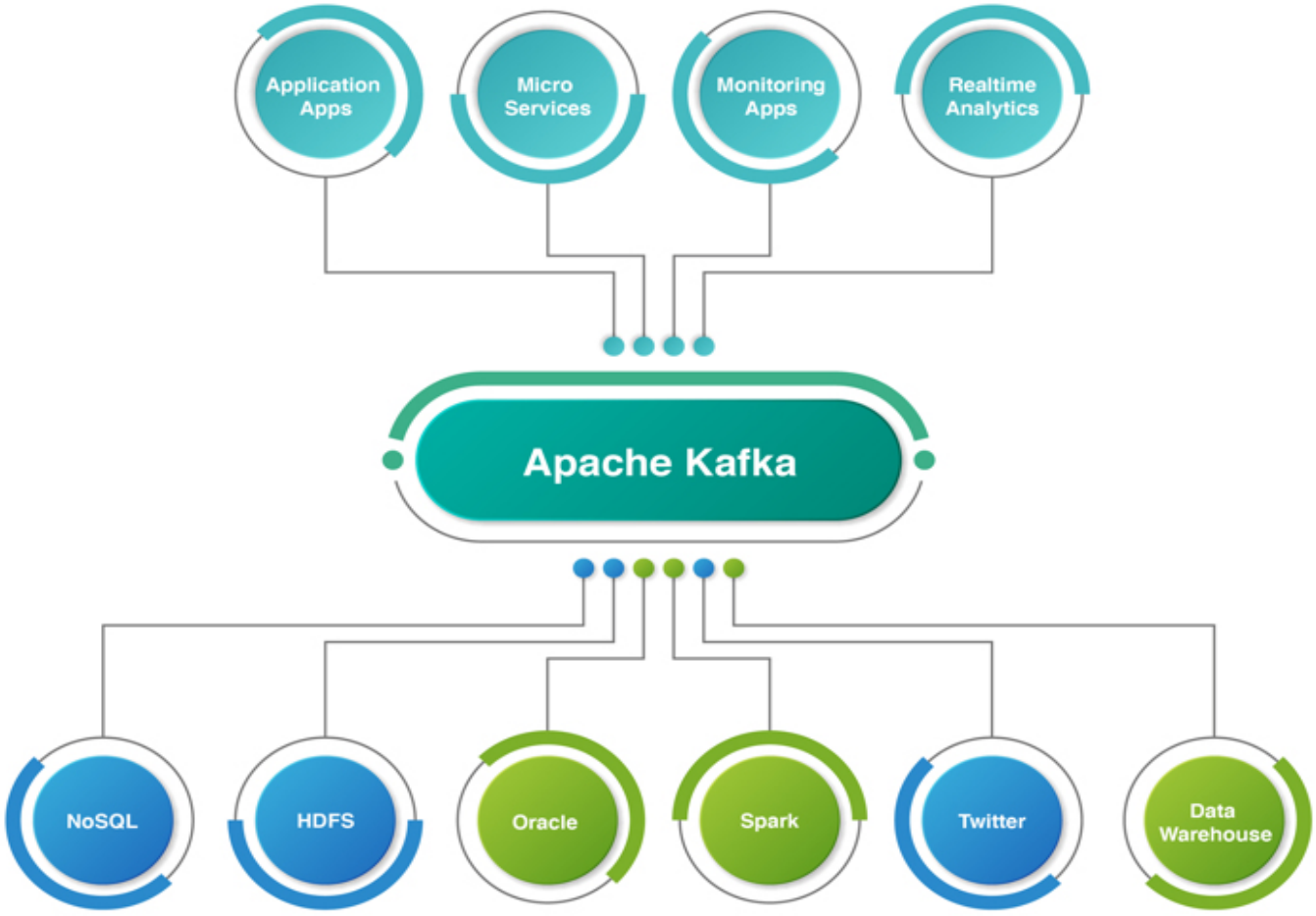
Apache Kafka e Spring Boot

Comunicação assíncrona entre microsserviços



Casa do Código | alura

EDUARDO FELIPE ZAMBOM SANTANA

Sumário

- ISBN
- Sobre o Livro
- Para quem é este livro?
- Sobre o autor
- Agradecimentos
- 1. Introdução
- 2. Instalando o Kafka
- 3. Criando uma API REST Spring Boot
- 4. Produzindo mensagens
- 5. Consumidor
- 6. Finalizando a shop-api
- 7. Diferentes grupos de consumidores
- 8. Paralelizando tarefas
- 9. Usando chaves nas mensagens
- 10. Retentativas
- 11. Administrando o Kafka no Java
- 12. Conectando no Kafka com Python
- 13. Configurações do Kafka
- 14. Executando todas as aplicações com o Docker
- 15. Testes de unidade
- 16. Kafka Streams

ISBN

Impresso: 978-65-86110-98-2

Digital: 978-65-86110-97-5

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Sobre o Livro

Normalmente quando precisamos fazer a comunicação entre microsserviços, logo pensamos em utilizar a comunicação síncrona, fazendo uma chamada HTTP de um serviço para outro. Esse tipo de implementação funciona na maioria dos casos, mas pode haver funcionalidades que podem causar diversos problemas. Por exemplo, imagine que queremos chamar um serviço que pode demorar alguns segundos até alguns minutos para responder. Se utilizarmos uma implementação síncrona, essa comunicação será bastante lenta e, se a aplicação não estiver preparada para isso, pode causar vários problemas como *timeouts* e retentativas de chamadas, o que pode piorar a situação da aplicação.

Para esse caso, a utilização de comunicação assíncrona pode ser interessante. Nela, informamos que queremos realizar um processamento, mas não esperamos que a resposta seja enviada na hora, e sim ficamos esperando uma notificação de que o processamento foi realizado ou que alguma falha aconteceu.

Uma forma bastante utilizada para essa implementação são as filas ou tópicos, na qual um processo adiciona uma mensagem na fila ou tópico informando que um processamento deve ser feito, e outro recebe essa mensagem quando estiver disponível e faz o processamento. Normalmente, o primeiro processo é conhecido como produtor, e o segundo, como consumidor.

Existem diversas funcionalidades que podem ser implementadas com processamento assíncrono, por exemplo: para processamentos demorados, para a comunicação com aplicações de terceiros e para a implementação de funcionalidades que não estão no fluxo principal da aplicação.

O Kafka é atualmente um dos principais sistemas para a implementação de processamento assíncrono disponível, pois ele, além de disponibilizar os tópicos para a produção e consumo de mensagens, também tem diversas funcionalidades que facilitam

bastante a implementação de nossas aplicações, como a distribuição e o balanceamento de carga entre os consumidores e o processamento de fluxos de dados.

O objetivo principal deste livro é mostrar como implementar aplicações utilizando o Kafka para fazer a comunicação assíncrona entre microsserviços implementados com o Spring Boot. Implementaremos uma aplicação completa, que vai ser incrementada capítulo a capítulo, começando pelos conceitos mais básicos como a criação dos tópicos e a produção e consumo de mensagens. Depois mostraremos a utilização das partes mais complexas do Kafka, como a utilização de grupos de consumidores para a distribuição e balanceamento de carga das mensagens, o mecanismo de retentativas quando houver falhas no processamento das mensagens e o processamento de fluxo de dados. Também apresentaremos algumas implementações interessantes como um consumidor implementado em Python, a configuração do Kafka na aplicação e a criação de testes de unidade para o código que envolve o Kafka.

Para quem é este livro?

Este livro foi escrito para pessoas programadoras que desejam entender e utilizar Kafka para a implementação da comunicação assíncrona entre microsserviços. Os exemplos de código foram implementados utilizando o Spring Boot, mas mesmo programadores de outros frameworks e outras linguagens podem utilizar o livro, já que o funcionamento do Kafka é o mesmo para qualquer linguagem de programação utilizada, mudando apenas a implementação.

Quando precisamos implementar a comunicação entre microsserviços, quase sempre já pensamos em utilizar a comunicação síncrona, mas veremos neste livro que a comunicação assíncrona pode ser bastante útil em diversos casos de uso, evitando problemas que poderiam ser causados em chamadas HTTP. Portanto, é importante que desenvolvedores e desenvolvedoras entendam como funciona a comunicação assíncrona.

Código-fonte

Todo o código-fonte das aplicações e os arquivos para a configuração do Docker estão disponíveis no GitHub, no repositório: https://github.com/ezambomsantana/livro-kafka

Sobre o autor

Eduardo Felipe Zambom Santana tem mais de 15 anos de experiência em Engenharia de Software. Trabalha principalmente com Java, já tendo trabalhado com os principais frameworks da linguagem como Struts, JSF e Spring. Também tem bastante experiência em outras linguagens como Python, Golang e Erlang. Formou-se em Ciência da Computação na UFSCar (2007), fez mestrado também na UFSCar (2010) e doutorado na USP (2019) trabalhando na área de sistemas distribuídos.

Agradecimentos

Agradeço à minha família, sem eles dificilmente teria chegado até aqui, e agradeço à minha namorada Brianda pelo companheirismo durante toda a minha carreira.

Obrigado ao pessoal da editora Casa do Código, em especial à Sabrina pela grande ajuda na revisão do livro.

Capítulo 1
Introdução

Quando interagimos com sistemas computacionais, na maioria das vezes, realizamos operações síncronas. Isso quer dizer que, quando executamos uma operação em um sistema, temos a resposta quase instantaneamente, seja a esperada ou não. Por exemplo, quando realizamos um saque de nossa conta corrente em um caixa eletrônico, recebemos o dinheiro e ele é descontado do saldo da nossa conta na hora. Também, quando entramos em um sistema de busca, digitamos o texto que desejamos procurar e recebemos uma lista de resultados imediatamente.

No entanto, nem toda operação pode ser executada dessa forma. Um exemplo que talvez passe desapercebido pela maioria dos usuários da internet são as compras on-line. Atualmente, na maioria dos e-commerces, quando fazemos uma compra, ela é registrada, mas não é processada na hora - o que chamamos de operação assíncrona -, isto é, fazemos um pedido, recebemos um e-mail indicando que a compra foi registrada e somente depois de algum tempo, sem que tenhamos que fazer nenhuma outra operação, é que recebemos o resultado, neste caso, a confirmação ou a rejeição da compra.

O processamento síncrono é muito mais simples de implementar, e normalmente garante uma satisfação maior do usuário, já que ele sabe em tempo real se sua operação foi realizada ou não. Porém, existem diversas situações em que é muito arriscado usar processamento síncrono, ou é simplesmente impossível implementar dessa forma. Algumas situações em que o processamento assíncrono é recomendável:

- Processamentos demorados: imagine que temos um sistema que processa milhões de registros em um banco de dados e queremos gerar um relatório com todas as transações diárias de

um grande banco. A geração desse relatório pode demorar minutos ou até horas. O usuário não ficará na frente do computador esperando esse relatório chegar, além disso, é possível acontecer um *timeout* na aplicação ou na rede, e o usuário perder a conexão com a aplicação. Neste caso, o melhor é que o usuário apenas peça a geração do relatório e depois de algum tempo receba uma notificação de que seu relatório está pronto.

- Integração com terceiros: em diversos sistemas, temos que fazer chamadas a serviços de terceiros, como em e-commerces, que, para aceitar pagamentos em cartão de crédito, devem chamar um serviço de processamento de pagamentos. Esse tipo de sistema pode demorar para responder, ou pode ser que ele mesmo seja assíncrono, então é bastante arriscado tentar implementar um sistema de confirmação de compras síncrono, pois se o serviço da operadora de cartão estiver com problemas, o e-commerce não conseguirá finalizar uma compra.

- Processamentos internos: toda aplicação faz diversos processamentos que não são a parte principal da aplicação, como a geração de logs de acesso. Se fizermos uma chamada síncrona para isso e o sistema que salva os logs da aplicação tiver uma lentidão inesperada, teremos uma lentidão no sistema principal por um requisito que não é importante para o fluxo principal da aplicação.

Obviamente, o processamento assíncrono possui diversos problemas difíceis de resolver, como dificuldade para depurar problemas e aumento da complexidade. É importante destacar que esse tipo de processamento não substitui o processamento síncrono, mas é um importante complemento. Existem diversas formas de fazer o processamento assíncrono, sendo o uso de sistemas gerenciadores de filas a principal delas. Basicamente, neste tipo de sistema, existe uma parte do código que gera uma mensagem e a coloca em uma

fila, e outra parte que recebe essa mensagem da fila para ser processada.

Por exemplo, no sistema de e-commerce que eu já comentei, após o usuário confirmar a compra, o sistema que recebeu a compra apenas a insere em uma fila e retorna para o usuário que a compra foi aceita. Se a fila de compras estiver pequena, praticamente instantaneamente a compra já será processada, porém, se estivermos em um período de muitas compras, como o Natal ou a Black Friday, essa fila pode estar grande, e a compra pode demorar alguns minutos para ser processada. Existem diversos sistemas de fila atualmente, todos com vantagens e desvantagens, como o ActiveMQ, o RabbitMQ e o Kafka. Neste livro, veremos como implementar esse tipo de aplicação com Kafka.

1.1 Apache Kafka

O Apache Kafka é uma ferramenta para o processamento de fluxos de eventos que são representados na forma de mensagens. O Kafka recebe uma mensagem indicando um evento, por exemplo, sobre uma nova compra que foi efetuada, e ele a encaminha para todos os interessados em recebê-la. Neste exemplo, já temos três dos principais conceitos dos sistemas de fila, que são:

- Produtor: sistema que gera um evento; no caso de um e-commerce, um serviço REST pode receber uma chamada para o registro de nova compra e esse serviço enviará uma mensagem para o Kafka indicando que esse evento ocorreu.
- Tópico: o Kafka pode possuir vários tópicos, que são formas de separar as mensagens e, principalmente, agrupar quem vai receber cada uma das mensagens. Sempre que um produtor envia uma mensagem, ele deve indicar para qual tópico do Kafka a mensagem está sendo enviada.

- Consumidor: sistema que recebe a mensagem do Kafka. Um consumidor sempre se inscreve em um tópico e recebe, por meio do Kafka, as mensagens enviadas para esse tópico.

Essas três características são comuns a praticamente todos os sistemas de fila, então qual a vantagem do Kafka sobre os outros sistemas? Existem várias vantagens, algumas das quais são compartilhadas com outras ferramentas e todas ficarão mais claras no decorrer dos capítulos, mas podemos adiantar uma breve descrição das principais:

- Durabilidade: o Kafka armazena toda a sua configuração e os dados em disco, fazendo com que não sejam perdidos mesmo com a reinicialização do sistema, seja em uma finalização normal ou por causa de algum erro.

- Paralelismo: o Kafka permite que diferentes processos se conectem a um mesmo tópico, o que possibilita que diferentes processos recebam as mesmas mensagens. Por exemplo, em um e-commerce queremos que um processo envie uma notificação para o usuário de que a compra foi cadastrada, e que outro faça a cobrança de cartão de crédito. O Kafka permite que os dois processos recebam as mensagens paralelamente de forma automática.

- Balanceamento de carga: o Kafka possui um mecanismo próprio para distribuir as mensagens em diferentes processos que estão inscritos no mesmo tópico, mas que não devem receber mensagens repetidas. Por exemplo, o processamento de cartão de crédito está lento, então queremos que diversos processos façam essa operação, mas não podemos deixar que uma mesma compra seja cobrada duas vezes.

- Streams: o Kafka possui suporte ao processamento de fluxos contínuos de dados nativo, incluindo transformações e operações como agregação e junções.

Toda mensagem no Kafka é formada por três valores: uma chave de identificação, um valor que são os dados passados na mensagem, e um *timestamp* da hora em que a mensagem foi salva no tópico.

```
Key: "compra-1"

Value: "Comprador: Eduardo, CPF: 123, Valor: 500"

Timestamp: "Jun. 25, 2020 at 2:06 p.m."
```

Neste exemplo, o valor é uma String simples, mas podemos enviar mensagens mais complexas, como dados no formato JSON, YAML ou XML, por exemplo. Esse suporte a qualquer padrão permite também que aplicações implementadas em diferentes linguagens consigam acessar um mesmo tópico do Kafka.

1.2 Spring Boot

Nos exemplos de código deste livro, utilizaremos o Spring Boot, um framework para facilitar o desenvolvimento de aplicações com o Spring, que hoje é o principal framework da linguagem Java. Ele possui diversas funcionalidades, como desenvolvimento de APIs REST e acesso a banco de dados. Além disso, ele possui uma implementação de um cliente da API do Kafka para a produção e consumo de mensagens e também para o gerenciamento dos tópicos, permitindo a criação e a exclusão de tópicos, por exemplo. Neste livro, utilizaremos quatro componentes principais do Spring Boot:

- O `spring-starter-web`, que será necessário para criar uma API REST que será responsável por receber os dados que serão colocados no Kafka e também por mostrar os dados depois do processamento para o usuário.

- O `spring-kafka-client`, que será utilizado para receber e enviar mensagens para um tópico do Kafka.
- O `spring-data-jpa`, que será utilizado para salvar algumas informações em um banco de dados H2.
- O `spring-boot-starter-test`, que será utilizado para o desenvolvimento de testes de unidades para nossas aplicações.

Além disso, para demonstrar a possibilidade da comunicação de serviços implementados em diferentes linguagens, implementaremos um consumidor em Python que se comunicará com a aplicação desenvolvida com o Spring Boot.

1.3 Python

Para mostrar que o Kafka é independente de linguagem, isto é, podemos conectar a um tópico do Kafka utilizando aplicações desenvolvidas em diferentes linguagens, também vamos implementar um consumidor de um tópico com Python. Assim como para Java, também existe uma biblioteca pronta, possibilitando uma conexão fácil ao Kafka utilizando essa linguagem.

Essa aplicação utilizará duas bibliotecas do Python, o `Flask` para a implementação de uma API REST e o `kafka-python`, que, como o próprio nome já diz, disponibiliza classes para acessar o Kafka.

1.4 Outras ferramentas

Utilizaremos algumas outras ferramentas no desenvolvimento das aplicações que serão mostradas neste livro.

Precisaremos utilizar uma IDE para desenvolvimento das aplicações. Eu fiz toda a implementação com o **IntelliJ**, mas qualquer outra IDE, como o VSCode, o Eclipse ou o NetBeans também pode ser utilizada. Como utilizaremos o Maven, qualquer uma dessas IDEs podem importar o projeto facilmente.

O **Maven** será utilizado para a configuração das dependências nas aplicações. A utilização dessa ferramenta é bastante simples: em todos os projetos, devemos criar um arquivo xml que possui todas as dependências que utilizaremos em nossa aplicação, e com isso ela faz o download e configura a aplicação para usar essas dependências. O Maven pode ser utilizado diretamente na IDE. Outra ferramenta que poderia ser utilizada para isso é o Gradle, mas os exemplos do livro usarão o Maven.

Para o desenvolvimento da aplicação Python, utilizei o **Visual Studio Code**.

O **Postman** será utilizado para fazer as chamadas para as APIs que serão criadas nas aplicações. Ele pode ser baixado em https://www.postman.com/. A versão gratuita da ferramenta tem todas as funcionalidades que precisaremos. No repositório do GitHub do projeto existe uma coleção do Postman com uma chamada de exemplo para todas as rotas das aplicações que serão desenvolvidas no livro.

O **H2** é um banco de dados em memória que é bem fácil de usar com aplicações Spring Boot. Veremos que é praticamente automática a utilização desse banco de dados, sem que nenhuma configuração tenha que ser feita. Utilizaremos esse banco de dados para armazenar os dados em algumas aplicações que serão desenvolvidas.

Utilizaremos o **Docker** para a criação de contêineres das aplicações que vamos desenvolver, bem como o **docker-compose**, para executar todas as aplicações e o Kafka de uma forma mais simples com os contêineres das aplicações.

1.5 Versões do Kafka, Spring Boot e Python

A primeira versão deste livro foi escrita com base na versão 2.8.0 do Kafka, lançada em abril de 2021. Esta segunda edição, que você está lendo agora, utiliza como base a versão 3.8.0, lançada em julho de 2024. Embora a estrutura básica da ferramenta permaneça a mesma e todo o código desenvolvido na primeira edição do livro continue funcionando normalmente, o Kafka passou por mudanças significativas, incluindo novas funcionalidades, melhorias de desempenho e segurança, além de ajustes que facilitam seu uso.

A mudança mais significativa é, talvez, a remoção gradual do ZooKeeper, anteriormente usado pelo Kafka para gerenciar o cluster. Atualmente, um cluster Kafka pode ser configurado usando o Kafka Raft (KRaft), que substitui o ZooKeeper e oferece várias melhorias, tornando a configuração do Kafka mais simples e eficiente. O ZooKeeper ainda é suportado, mas sua remoção completa está prevista para a versão 4.0.0 do Kafka. Por isso, para novas instalações, recomenda-se o uso do KRaft, que é o que será feito nesta versão do livro.

A aplicação Spring Boot desenvolvida para demonstrar a utilização do Kafka foi desenvolvida com a versão 2.4.4 do framework e o Java 8. Nos últimos anos, tanto o Spring quanto o Java, tiveram uma grande evolução, na nova versão do livro vamos usar a versão 3.3.5 do Spring Boot e a versão 21 do Java. Apesar disso, poucas alterações serão feitas no código por causa da atualização das versões.

As únicas duas grades mudanças são os imports de classes do Java, que mudaram do pacote `javax` para `jakarta`, e pequenas alterações no SQL de algumas aplicações, pois a sintaxe do banco de dados H2, usados nas aplicações, foram alteradas.

Por último, a primeira versão do livro usou a versão 3.5 do Python na aplicação mostrada no capítulo 12, e na versão atual do livro será utilizada a versão 3.13.

Capítulo 2
Instalando o Kafka

Para instalar o Kafka em qualquer sistema operacional, basta fazer o download no site oficial da ferramenta e descompactar o arquivo (https://kafka.apache.org/). Para iniciar o Kafka no Linux e no Mac, os passos são exatamente os mesmos: primeiro definimos o id do cluster do Kafka e adicionamos essa configuração nos arquivos do Kafka:

```
KAFKA_CLUSTER_ID="$(bin/kafka-storage.sh random-uuid)"
./bin/kafka-storage.sh format -t $KAFKA_CLUSTER_ID -c
config/kraft/server.properties
```

> **Windows**
>
> Para usuários do Linux e do MacOS, os comandos são exatamente iguais aos que estão no livro, para usuários do Windows há uma pequena diferença. Nesse sistema operacional, os scripts para execução do Kafka estarão na pasta `/bin/windows/` e, em vez da extensão `.sh`, eles terão a extensão `.bat`. Isso é válido para todos os comandos que serão executados neste capítulo. Então, em todos os comandos, basta alterar o caminho do comando para adicionar o `/windows/` e mudar a extensão do arquivo.

Esses dois comandos precisam ser executados uma única vez, na primeira inicialização do Kafka. Depois disso, podemos executar o comando que efetivamente inicializa o Kafka:

```
./bin/kafka-server-start.sh config/kraft/server.properties
```

Veja que nesse comando tivemos que passar o caminho para o arquivo de configuração do Kafka, o `config/kraft/server.properties`. Inicialmente vamos utilizar as configurações padrões, mas nos

próximos capítulos vamos analisar algumas das propriedades que podem ser mudadas neste arquivo. Além disso, usando o Spring Boot, poderemos definir configurações específicas para a nossa aplicação para sobrescrever as configurações padrões do Kafka.

Depois de executar esse comando, devem aparecer várias mensagens no console, que indicam os passos que são executados para a inicialização da ferramenta, e também são escritas algumas das suas principais configurações.

Como explicado final do capítulo anterior, as versões antigas do Kafka usavam o ZooKeeper para o gerenciamento do cluster, por isso antigamente também era necessário inicializar o ZooKeeper junto do Kafka. Isso não é mais necessário nas versões atuais do Kafka, quando utilizado o KRaft.

2.1 Tópicos

Como falado na introdução, o conceito mais importante são os tópicos. São neles onde adicionamos e lemos as mensagens que as aplicações vão compartilhar. Agora que já temos o Kafka sendo executado em nossa máquina, podemos fazer alguns testes e realizar operações com os tópicos. Vamos primeiro criar um tópico utilizando o comando:

```
./bin/kafka-topics.sh \
    --create \
    --topic topico-teste \
    --bootstrap-server localhost:9092
```

Neste comando, utilizamos o `kafka-topics`, que é uma ferramenta para gerenciar os tópicos do Kafka, com a qual podemos listar, criar e excluir tópicos. O parâmetro `--create` indica que vamos criar um novo tópico; o `--topic` serve para definir o nome do tópico, no nosso caso, `topico-teste` e, no `--bootstrap-server`, deve ser passado

o endereço e a porta que o Kafka está rodando, no nosso caso `localhost:9092`. Note que essa ferramenta é independente da instalação do Kafka. Se passarmos um endereço diferente no `--bootstrap-server`, ela se conectará a outra instância do Kafka.

Além de criar os tópicos, a ferramenta `kafka-topics` também possui a funcionalidade de listar tópicos. Para isso, podemos executar o comando:

```
./bin/kafka-topics.sh \
    --list \
    --bootstrap-server localhost:9092
```

No nosso caso, esse comando retornará o único tópico que criamos por enquanto, que é o `topico-teste`.

Outro comando interessante é o que retorna detalhes de um tópico específico, que é o `describe`, onde passamos o tópico cujas informações serão retornadas. O comando a ser executado é:

```
./bin/kafka-topics.sh \
    --describe \
    --topic topico-teste \
    --bootstrap-server localhost:9092
```

O retorno desse comando será um conjunto de informações sobre esse tópico como mostrado a seguir:

```
Topic: topico-teste    TopicId: om2YlC2yQrSKraJZ1zk_gA    PartitionCount:
1
ReplicationFactor: 1    Configs: segment.bytes=1073741824
    Topic: topico-teste    Partition: 0    Leader: 0    Replicas: 0
Isr: 0
```

Vamos entender o que todas essas informações sobre os tópicos significam nos próximos capítulo. Agora que o Kafka já está rodando, podemos começar a utilizar esse tópico para produzir e consumir mensagens nele.

2.2 Enviando e recebendo mensagens do Kafka

Assim como o Kafka tem uma ferramenta para gerenciar os tópicos, existe uma para enviar mensagens para o Kafka, a `kafka-console-producer` . Podemos utilizá-la para enviar mensagens diretamente para a nossa instância do Kafka ou também para uma instância remota. Para executá-la, basta rodar o comando:

```
./bin/kafka-console-producer.sh \
    --topic topico-teste \
    --bootstrap-server localhost:9092
```

Esse comando iniciará um produtor de mensagens, cada mensagem digitada será enviada para o tópico. Os parâmetros do comando são os mesmos, o `--topic` indica para qual tópico queremos mandar as mensagens, e o `--bootstrap-server` indica para qual instância do Kafka a mensagem será enviada.

Agora que temos um produtor, podemos também inicializar um ou mais consumidores. Para isso, usamos a ferramenta `kafka-console-consumer` , que imprimirá na tela todas as mensagens recebidas por um tópico. Para executar essa ferramenta, basta executar o comando:

```
./bin/kafka-console-consumer.sh \
    --topic topico-teste \
    --bootstrap-server localhost:9092
```

Se enviarmos três mensagens para o tópico `topico-teste` , a ferramenta imprimirá as mensagens na sequência correta:

```
eduardo@eduardo:~/dev/kafka_2.13-2.8.0$ bin/kafka-console-consumer.sh --
topic topico-teste --bootstrap-server localhost:9092
mensagem1
mensagem2
mensagem3
```

Se você já tinha enviado algumas mensagens para o Kafka antes de inicializar o consumidor, deve ter percebido que essas mensagens

não foram lidas. Isso porque, por padrão, o consumidor só começa a receber mensagens que foram enviadas depois que ele é criado. Para alterar esse comportamento, podemos adicionar a flag `--from-beginning`, que indicará para o consumidor que todas as mensagens que existem no tópico devem ser lidas. O comando ficará assim:

```
./bin/kafka-console-consumer.sh \
    --topic topico-teste \
    --from-beginning \
    --bootstrap-server localhost:9092
```

2.3 Consumer Groups

No Kafka é possível definir grupos de consumidores, que servem para definir como será a distribuição de mensagens entre os consumidores de um tópico. Por exemplo, se quisermos que dois consumidores leiam todas as mensagens enviadas para um mesmo tópico, basta alocar os dois consumidores em grupos diferentes. Assim, o Kafka fará sozinho o controle de quais mensagens cada um dos consumidores já processou. Sempre que iniciamos um consumidor, podemos definir em qual grupo ele está:

```
./bin/kafka-console-consumer.sh \
    --topic topico-teste \
    --from-beginning \
    --bootstrap-server localhost:9092 \
    --consumer-property group.id=grupo-1
```

Note que foi adicionada a opção `--consumer-property group.id=grupo-1` ao comando de inicialização do consumidor, ela indica que esse consumidor fará parte do grupo chamado `grupo-1`. Podemos criar outro consumidor agora, que também receberá as mensagens do

mesmo tópico, mas podemos definir que ele está em outro grupo, no caso o `grupo-2`.

```
./bin/kafka-console-consumer.sh \
    --topic topico-teste \
    --from-beginning \
    --bootstrap-server localhost:9092 \
    --consumer-property group.id=grupo-2
```

Se executarmos o produtor agora e enviarmos mensagens para o tópico `topico-teste`, veremos que os dois consumidores receberão todas as mensagens enviadas para o tópico. Isso será importante se tivermos duas aplicações que precisam consumir os dados de um mesmo tópico. Por exemplo, em aplicação de e-commerce, podemos ter uma aplicação que consome todas as compras para fazer o processamento do cartão de crédito, e outra que processa as compras para enviar um e-mail para o comprador avisando que a compra foi cadastrada e está sendo processada.

Podemos também listar os *Consumer Groups* que foram criados no Kafka até agora com a ferramenta `kafka-consumer-groups`. A listagem a seguir mostra como executar essa ferramenta.

```
./bin/kafka-consumer-groups.sh \
    --bootstrap-server localhost:9092 \
    --list
```

O resultado desse comando será:

```
grupo-2
grupo-1
```

Os grupos são um dos mecanismos mais poderosos do Kafka, eles permitem que façamos o balanceamento de carga e a distribuição de mensagens sem praticamente termos que implementar nenhuma linha de código. Nos capítulos 7 e 8 será explicado detalhadamente como esses grupos funcionam e como utilizá-los em uma aplicação com o Spring Boot.

2.4 Parando o Kafka

Parar o Kafka é simples: podemos apenas finalizar a execução do comando que iniciou o Kafka no início deste capítulo. Porém, os dados do Kafka são permanentes. Se pararmos e reinicializarmos o Kafka, veremos que os tópicos e as mensagens enviadas continuarão existindo como se nada tivesse acontecido, o que garante que o Kafka possa ser reinicializado depois de uma falha sem que haja perda de mensagens. Se quisermos reinicializar totalmente a nossa instância do Kafka, temos que excluir os arquivos que foram criados durante a execução do Kafka.

Os diretórios onde os arquivos do Kafka são salvos são configurados no arquivo de propriedades `config/kraft/server.properties`. Por padrão, eles são salvos no caminho `/tmp/kraft-combined-logs`. Podemos alterar esse diretório modificando a configuração padrão do Kafka.

Se quisermos reinicializar totalmente o Kafka, basta excluir essa pasta com o comando:

```
rm -rf /tmp/kraft-combined-logs
```

Já temos agora o Kafka rodando em nossa máquina, mas apenas enviamos mensagens usando as ferramentas do próprio Kafka. Nos próximos capítulos, vamos começar a utilizar o Kafka em nossas aplicações Java. Veremos que podemos produzir e enviar mensagens nos tópicos da mesma forma que fizemos neste capítulo. Também veremos que o Kafka disponibiliza diversas outras funcionalidades que podem facilitar bastante o desenvolvimento de nossas aplicações, mas isso fica para os próximos capítulos.

CAPÍTULO 3
Criando uma API REST Spring Boot

Para utilizar o Kafka, vamos precisar de dados de entrada e de alguma forma de mostrar os dados de saída para o usuário. A melhor maneira de fazer isso é utilizando uma API REST. Neste capítulo, utilizando o Spring Boot, vamos mostrar como criar uma API simples que receberá os dados de uma compra e as salvará em um banco de dados H2. A partir dos próximos capítulos, esses dados de entrada serão enviados para um tópico do Kafka.

Para o desenvolvimento da API, vamos precisar de uma IDE. Eu utilizei o IntelliJ, mas qualquer outra como Eclipse ou o VSCode pode ser utilizada também. Além disso, utilizei o Maven para o gerenciamento das dependências, mas, caso você prefira, também é possível utilizar o Gradle.

O projeto consiste basicamente em uma API para o cadastro de compras, na qual teremos duas rotas. A primeira, para cadastrar as compras, receberá uma lista de produtos com a quantidade e o preço do produto e salvará a compra e seus itens no banco de dados. A segunda rota apenas listará todas as compras cadastradas. O nome dessa aplicação será `shop-api`, já que ela possuirá a API para que um usuário efetue as compras em nosso sistema.

3.1 Configuração do projeto

O primeiro passo é criar um projeto Maven em sua IDE. Depois, no arquivo `pom.xml`, vamos adicionar o Spring Boot e duas dependências ao projeto: o `spring-boot-starter-web` e o H2. A primeira dependência é utilizada para a criação da API e a segunda é o banco de dados que vamos utilizar em nossa aplicação. O H2 é

um banco em memória bastante útil e fácil de usar. Adicionando-o ao projeto, ao subir a aplicação, o banco de dados será inicializado e estará pronto para ser utilizado.

> **SPRING INITIALIZR**
>
> Neste exemplo, eu vou criar e configurar a aplicação Spring diretamente na IDE, mas outra possibilidade é usar o Spring Initializr (https://start.spring.io/). Essa ferramenta oferece um ambiente visual para configurar uma aplicação Spring, sendo possível configurar a versão do Spring, a versão do Java e adicionar as dependências iniciais do projeto. Depois de criado, basta fazer o download do projeto e abri-lo na sua IDE de preferência.

Também utilizaremos o *Lombok*, uma biblioteca para adicionar automaticamente diversos métodos que tem praticamente sempre a mesma implementação, como os `get` e `set`, `toString` e construtores. Ele é simples de usar, basta adicionar algumas anotações à classe. Para adicionar essa biblioteca, temos que adicionar a biblioteca `org.projectlombok` ao `pom.xml`. Sempre que alguma anotação do Lombok for utilizada, eu explicarei o que ela faz. Para saber mais detalhes, acesse o site oficial da ferramenta em https://projectlombok.org/. O código a seguir mostra o arquivo `pom.xml` do Maven com as configurações citadas, e também a dependência do `lombok`.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.santana</groupId>
    <artifactId>shop-api</artifactId>
```

```
    <version>1.0-SNAPSHOT</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.3.5</version>
    </parent>

    <properties>
        <maven.compiler.source>21</maven.compiler.source>
        <maven.compiler.target>21</maven.compiler.target>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.34</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>

</project>
```

3.2 Banco de dados e Spring Boot

Para a configuração de algumas propriedades do Spring Boot, vamos criar o arquivo `aplication.properties` dentro da pasta `src/main/resources`. Esse arquivo contém diversas configurações da aplicação, das quais utilizaremos duas. A primeira é o `server.port`, que define a porta em que a aplicação será executada. A porta padrão do Spring Boot é a 8080, então não precisaríamos definir essa configuração aqui, mas nas próximas aplicações utilizaremos portas diferentes, então vou deixar explícito qual porta utilizaremos em todas as aplicações. A outra configuração é a `spring.jpa.hibernate.ddl-auto` com valor `none`, o que indica para o Spring Boot não criar as tabelas baseadas nas classes do modelo (que veremos a seguir), isso porque definiremos os scripts para a criação das tabelas explicitamente. A listagem a seguir mostra o conteúdo do arquivo `application.properties`.

```
server.port=8080
spring.jpa.hibernate.ddl-auto=none
```

O banco de dados da aplicação terá duas tabelas, a `shop`, na qual serão armazenadas as compras, e a `shop_item`, que armazenará os itens da compra. O script a seguir tem os comandos SQL para a criação das tabelas. Para que o Spring crie automaticamente as tabelas, toda vez que executamos a aplicação devemos colocar um arquivo sql com o script na pasta `src/main/resources`. O arquivo deve ser chamado de `schema.sql` - esse nome é obrigatório, já que o Spring Boot buscará um arquivo com esse nome para a criação das tabelas.

Os campos da tabela `shop` são o `id`, que será gerado pelo próprio banco de dados; o `identifier`, que será um UUID gerado na aplicação antes que a compra seja salva no banco de dados; o `status` da compra, que poderá ser `PENDING`, `SUCCESS` e `ERROR`; e a data em que a compra foi efetuada.

Os campos da tabela `shop_item` são o `id`, também gerado pelo banco de dados; o `product_identifier`, que é o identificador do produto que será enviado pelo usuário; o `amount`, que terá a quantidade de itens comprados; o `price`, que terá o preço dos itens; e o `shop_id`, que é a chave estrangeira para associar o item à compra.

```
create table shop (
    id bigint generated by default as identity,
    identifier varchar not null,
    status varchar not null,
    date_shop date,
    primary key (id)
);

create table shop_item (
    id bigint generated by default as identity,
    product_identifier varchar(100) not null,
    amount int not null,
    price float not null,
    shop_id bigint REFERENCES shop(id),
    primary key (id)
);
```

Toda aplicação do Spring Boot deve ter uma classe que possui o método `main`, que será sempre responsável pela inicialização da aplicação. Essa classe pode ter qualquer nome (eu a chamei de `Main`) e deve ter também a anotação `@SpringBootApplication`.

```
package com.santana;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Main {

    public static void main(String [] args) {
        SpringApplication.run(Main.class, args);
```

```
    }

}
```

Para executar a aplicação, basta executar essa classe que possui o método `main` utilizando qualquer IDE, basta clicar com o botão direito sobre essa classe e mandar executar. Do jeito que está, já é possível executar a aplicação. Como ainda não temos nenhuma rota criada, a aplicação não fará nada ainda, mas já será possível visualizar o log de uma aplicação Spring Boot sendo executada.

3.3 Modelo e repositório

Agora que a configuração está pronta, podemos começar a implementação da aplicação. Vamos começar com a camada do modelo. Essa camada possui classes que representam os objetos que serão armazenados e recuperados do banco de dados. Por exemplo, se temos a tabela `shop`, teremos também uma classe `Shop`, que possuirá os mesmos valores que armazenarmos nas tabelas. As classes do modelo podem ser chamadas também de *entidades*.

Na nossa aplicação, na camada do modelo, vamos precisar de duas classes que representam as duas tabelas que temos no banco de dados. A primeira é a `ShopItem`, que representará os itens da compra do usuário. Como essa classe é uma entidade do banco de dados, precisamos colocar a anotação `@Entity` indicando o nome da tabela.

Essa classe possui exatamente os mesmos atributos da tabela, que são o `productIdentifier`, o `amount` e o `price`. Além disso, existe o relacionamento indicando a qual compra esse item pertence. Essa classe tem duas anotações do `lombok`, a `@Getter` e a `@Setter`, que geram automaticamente no código os métodos `get` e `set` para todos os atributos da classe.

O código a seguir mostra a implementação da classe `ShopItem`.

```
package com.santana.model;

import lombok.Getter;
import lombok.Setter;

import jakarta.persistence.*;

import com.santana.dto.ShopItemDTO;

@Getter
@Setter
@Entity(name = "shop_item")
public class ShopItem {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "product_identifier")
    private String productIdentifier;

    private Integer amount;

    private Float price;

    @ManyToOne
    @JoinColumn(name = "shop_id")
    private Shop shop;

    public static ShopItem convert(ShopItemDTO shopItemDTO) {
        ShopItem shopItem = new ShopItem();
        shopItem.setProductIdentifier(
            shopItemDTO.getProductIdentifier());
        shopItem.setAmount(shopItemDTO.getAmount());
        shopItem.setPrice(shopItemDTO.getPrice());
        return shopItem;
    }
```

```
}
```

Uma observação: o método `convert` ficará apontando erro enquanto os DTOs não forem criados. Se preferir, pode criar esse método após a criação dos DTOs.

A outra entidade é a `Shop`, que representa cada compra de um usuário. Ela também possui exatamente os mesmos campos da tabela `shop`, como `id`, `identifier`, `status` e o `dateShop`. O código a seguir mostra a implementação dessa entidade.

```
package com.santana.model;

import lombok.Getter;
import lombok.Setter;

import jakarta.persistence.*;

import com.santana.dto.ShopDTO;

import java.time.LocalDate;
import java.util.List;
import java.util.stream.Collectors;

@Getter
@Setter
@Entity
public class Shop {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String identifier;

    private String status;

    @Column(name = "date_shop")
```

```
    private LocalDate dateShop;

    @OneToMany(fetch = FetchType.EAGER,
        cascade = CascadeType.ALL,
        mappedBy = "shop")
    private List<ShopItem> items;

    public static Shop convert(ShopDTO shopDTO) {
        Shop shop = new Shop();
        shop.setIdentifier(shopDTO.getIdentifier());
        shop.setStatus(shopDTO.getStatus());
        shop.setDateShop(shopDTO.getDateShop());
        shop.setItems(shopDTO
                .getItems()
                .stream()
                .map(i -> ShopItem.convert(i))
                .collect(Collectors.toList()));
        return shop;
    }

}
```

O `JpaRepository` é uma interface do `spring-data` que já disponibiliza diversos métodos para a manipulação das entidades no banco de dados, como o `save`, `findAll` e o `findById`. Para utilizar esses métodos, basta criar uma nova interface, no nosso caso a `ShopRepository`, e estender a interface `JpaRepository`.

```
package com.santana.repository;

import com.santana.model.Shop;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface ShopRepository
    extends JpaRepository<Shop, Long> {
```

```
}
```

Também é possível implementar consultas utilizando `JPQL` ou `SQL` nativos. Neste capítulo, ainda não faremos nenhuma consulta complexa, por isso é o suficiente apenas criar a interface sem nenhum novo método.

3.4 DTOs e Controller

Não é uma boa prática retornar os objetos do modelo em APIs, pois o modelo tem grandes chances de mudar, o que afetaria o retorno dos dados da API. Além disso, pode haver dados que não interessam aos usuários, como ids, datas de criação e atualização de objetos. Por isso, podemos utilizar os DTOs (Data Transfer Object), que são classes que possuem apenas os dados que devem ser retornados na API. Normalmente eles são parecidos com as classes do modelo, mas isso não é obrigatório. Podemos também criar DTOs que retornam dados de mais de uma entidade.

No nosso exemplo, os DTOs terão os mesmos atributos que nas classes do modelo, exceto os ids do banco de dados, que é um dado que não interessa ao usuário da aplicação. Além disso, adicionaremos um método para converter um objeto do modelo em um DTO. No exemplo a seguir, temos a classe `ShopItemDTO`, que é o DTO da classe do modelo `ShopItem`. Note que, assim como no modelo, estamos utilizando as anotações `@Getter` e `@Setter` do `lombok`.

```
package com.santana.dto;

import com.santana.model.ShopItem;

import lombok.Getter;
import lombok.Setter;
```

```
@Getter
@Setter
public class ShopItemDTO {

    private String productIdentifier;
    private Integer amount;
    private Float price;

    public static ShopItemDTO convert(ShopItem shopItem) {
        ShopItemDTO shopItemDTO = new ShopItemDTO();
        shopItemDTO.setProductIdentifier(
            shopItem.getProductIdentifier());
        shopItemDTO.setAmount(shopItem.getAmount());
        shopItemDTO.setPrice(shopItem.getPrice());
        return shopItemDTO;
    }

}
```

Perceba que, no código a seguir, a classe `ShopDTO` é o DTO da classe do modelo `Shop`. Ela tem exatamente as mesmas características da classe do exemplo anterior.

```
package com.santana.dto;

import lombok.Getter;
import lombok.Setter;

import java.time.LocalDate;
import java.util.List;
import java.util.stream.Collectors;

import com.santana.model.Shop;

@Getter
@Setter
public class ShopDTO {
    private String identifier;
```

```
    private LocalDate dateShop;
    private String status;
    private List<ShopItemDTO> items = new ArrayList<>();

    public static ShopDTO convert(Shop shop) {
        ShopDTO shopDTO = new ShopDTO();
        shopDTO.setIdentifier(shop.getIdentifier());
        shopDTO.setDateShop(shop.getDateShop());
        shopDTO.setStatus(shop.getStatus());
        shopDTO.setItems(shop
                    .getItems()
                    .stream()
                    .map(i -> ShopItemDTO.convert(i))
                    .collect(Collectors.toList()));
        return shopDTO;
    }

}
```

A última parte da aplicação é o desenvolvimento do `@RestController`, que é a classe que disponibilizará as rotas da aplicação. No nosso exemplo, temos duas rotas, a `GET /shop`, que será utilizada para listar todas as compras que existem no banco de dados, e a `POST /shop`, que será utilizada para cadastrar uma nova compra.

A primeira rota, implementada no método `getShop`, chama o método `findAll` do repositório para listar todas as compras cadastradas e, depois, apenas mapeamos as classes do modelo para os DTOs. Já a segunda rota, implementada no método `saveShop`, recebe um DTO de uma compra, cria um novo UUID, salva a data de hoje nos campos `identifier` e `dateShop`, converte o DTO para um modelo e salva a compra no banco de dados chamando o método `save` do repositório.

Aqui utilizamos outra anotação do lombok, a `@RequiredArgsConstructor`, para fazer a injeção de dependência do `ShopRepository` na classe `ShopController`. Sem essa anotação, precisaríamos utilizar a anotação `@Autowired` do Spring Boot para

fazer a injeção de dependência em cada um dos atributos da classe. Como em nossa classe `ShopController` só temos um atributo, o `shopRepository`, só precisaríamos de um `@Autowired`, mas em uma classe com mais atributos teríamos que repetir essa anotação várias vezes.

```
package com.santana.controller;

import com.santana.dto.ShopDTO;
import com.santana.model.Shop;
import com.santana.model.ShopItem;
import com.santana.repository.ShopRepository;
import lombok.RequiredArgsConstructor;
import org.springframework.web.bind.annotation.*;

import java.time.LocalTime;
import java.util.List;
import java.util.UUID;
import java.util.stream.Collectors;

@RestController
@RequestMapping("/shop")
@RequiredArgsConstructor
public class ShopController {

    private final ShopRepository shopRepository;

    @GetMapping
    public List<ShopDTO> getShop() {
        return shopRepository
                .findAll()
                .stream()
                .map(shop -> ShopDTO.convert(shop))
                .collect(Collectors.toList());
    }

    @PostMapping
    public ShopDTO saveShop(@RequestBody ShopDTO shopDTO) {
        shopDTO.setIdentifier(
```

```
            UUID.randomUUID().toString());
        shopDTO.setDateShop(LocalDate.now());
        shopDTO.setStatus("PENDING");

        Shop shop = Shop.convert(shopDTO);
        for (ShopItem shopItem : shop.getItems()) {
            shopItem.setShop(shop);
        }

        return ShopDTO.convert(shopRepository.save(shop));
    }

}
```

Uma observação: não é uma boa prática chamar o repositório diretamente do controlador. Fizemos isso nesse exemplo inicial apenas por simplicidade, mas nos próximos capítulos implementaremos uma camada intermediária chamada Service.

3.5 Executando a aplicação

Executar a aplicação agora é bastante simples. Se você estiver utilizando alguma IDE, basta executar a classe com o método `main`. Caso vá rodar diretamente por linha de comando, pode executar utilizando o comando `mvn spring-boot:run`. Como estamos utilizando as configurações padrão do Spring Boot, a aplicação subirá na porta 8080.

Para chamar a rota que cria uma compra, temos que chamar o endereço `http://localhost:8080/shop` com o método `POST`, passando como parâmetro os dados de uma compra no formato JSON, como:

```
{
    "items": [
        {
            "productIdentifier": "123456789",
```

```
            "amount": "100",
            "price": "1000"
        },
        {
            "productIdentifier": "123456789",
            "amount": "100",
            "price": "1000"
        }
    ]
}
```

No retorno, serão retornados além da lista de itens, o `identifier`, a data e o status da compra, por exemplo:

```
{
    "identifier": "da7b1e40-2bcc-4c27-bdbb-953dcd7e8e7f",
    "dateShop": "20:08:09.043921",
    "status": "PENDING",
    "items": [
        {
            "productIdentifier": "123456789",
            "amount": 100,
            "price": 1000.0
        },
        {
            "productIdentifier": "123456789",
            "amount": 100,
            "price": 1000.0
        }
    ]
}
```

Para chamar a rota que lista as compras, temos que chamar o endereço `http://localhost:8080/shop` com o método `GET`. O retorno dessa rota será algo como:

```
[
    {
        "identifier": "da7b1e40-2bcc-4c27-bdbb-953dcd7e8e7f",
        "dateShop": "20:08:09",
```

```
    "status": "PENDING",
    "items": [
        {
            "productIdentifier": "123456789",
            "amount": 100,
            "price": 1000.0
        },
        {
            "productIdentifier": "123456789",
            "amount": 100,
            "price": 1000.0
        }
    ]
},
{
    "identifier": "0f310ae8-444f-4b59-8a9f-0358d8ff551b",
    "dateShop": "20:09:18",
    "status": "PENDING",
    "items": [
        {
            "productIdentifier": "123456789",
            "amount": 100,
            "price": 1000.0
        },
        {
            "productIdentifier": "123456789",
            "amount": 100,
            "price": 1000.0
        }
    ]
},
{
    "identifier": "521768bf-011d-487c-88fa-7cf046776c87",
    "dateShop": "20:09:19",
    "status": "PENDING",
    "items": [
        {
            "productIdentifier": "123456789",
            "amount": 100,
```

```
                "price": 1000.0
            },
            {
                "productIdentifier": "123456789",
                "amount": 100,
                "price": 1000.0
            }
        ]
    }
]
```

Este capítulo apresentou a implementação de uma API com o Spring Boot. Isso será importante porque será através dessa API que receberemos os dados que enviaremos para o Kafka. Também será através dela que apresentaremos os dados que serão processados pelos consumidores do Kafka. Então vamos continuar avançando porque estamos só no começo.

Capítulo 4
Produzindo mensagens

Temos a aplicação recebendo os dados de compra, mas agora temos que enviar a compra para ser validada e depois confirmada. No nosso caso, teremos que verificar se existe estoque para o produto comprado e, caso exista, vamos baixar o estoque com a quantidade comprada e enviar uma confirmação de que a compra foi confirmada. Caso não exista estoque, vamos enviar uma mensagem informando que a compra foi cancelada.

Poderíamos fazer isso diretamente no nosso serviço de compra, mas nesse caso teríamos que ter as informações de compra e estoque em apenas um serviço. Além disso, colocando os dados das compras em um tópico do Kafka, esses dados ficam disponíveis para diversos outros serviços, por exemplo, para geração de relatórios. Neste capítulo, além de salvar a compra no banco de dados, vamos enviá-la para um tópico Kafka. Para isso, teremos que modificar um pouco nossa aplicação original para adicionar o código que acessará o Kafka.

4.1 Configuração

Vamos criar o produtor utilizando o projeto que iniciamos no capítulo anterior, por isso, a única configuração que temos que fazer agora, é adicionar a dependência do Spring Boot que faz a conexão com o Kafka ao arquivo `pom.xml` . Para isso, basta adicionar as seguintes linhas às dependências:

```
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
</dependency>
```

Essa biblioteca `spring-kafka` terá todo o código de que necessitaremos para implementar tanto o produtor quanto o consumidor. Veremos que o Spring Boot disponibiliza classes que facilitam bastante o nosso trabalho, e basicamente com algumas configurações e poucas linhas de código poderemos utilizar os tópicos do Kafka.

4.2 Implementando o produtor

Para implementar o produtor, que é o código que gera a mensagem que será enviada para o Kafka, vamos precisar de duas classes novas que acessam o Kafka. A primeira, a `KafkaConfig`, é apenas para configurar o Kafka em nossa aplicação. Essa classe é uma configuração do Spring Boot, por isso a anotação `@Configuration`. Apesar de por enquanto essa aplicação ser apenas um produtor, já vamos configurar tanto a produção quanto o consumo de mensagens, pois utilizaremos essa configuração mais tarde.

Para a configuração do produtor, utilizamos o *bean* `kafkaTemplate`, que será o objeto usado para enviar as mensagens para o Kafka. Os `Beans` são os objetos do Spring que poderão depois ser utilizados em outras partes da nossa aplicação. Nesse caso, podemos utilizar o `kafkaTemplate` sempre que quisermos enviar uma mensagem para o Kafka.

O atributo `bootstrapAddress` definirá o endereço do Kafka que vamos acessar. Podemos definir o valor no arquivo `application.properties` ou utilizar um valor padrão, que é o que faremos por enquanto. Note que estamos utilizando o endereço `localhost:9092`, o localhost, porque acessaremos o Kafka que está rodando na nossa máquina mesmo, mas poderíamos também configurar o Kafka em outra máquina; e a porta 9092, pois é a porta padrão do Kafka, então se mudarmos a porta na configuração do Kafka também temos que mudar nesta parte do código da nossa aplicação.

No método `producerFactory`, definiremos diversas propriedades do acesso ao Kafka e veremos diferentes opções no decorrer dos capítulos. Aqui, vamos apenas definir três valores: o endereço do Kafka na propriedade `ProducerConfig.BOOTSTRAP_SERVERS_CONFIG`, o tipo da chave `ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG` (falaremos mais sobre chaves no capítulo 7), o `ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG`, no qual definimos que vamos mandar um `JSON` para o Kafka, e o `ProducerConfig.CLIENT_ID_CONFIG`, que define um identificador para o produtor.

No método `kafkaTemplate`, apenas definimos o bean `kafkaTemplate` utilizando as propriedades definidas no método `producerFactory`.

Para a configuração do consumidor, que será utilizado mais tarde, temos que definir o bean `kafkaListenerContainerFactory`. Neste método, também poderemos definir diversas configurações para o consumo das mensagens.

```
package com.santana.events;

import java.util.HashMap;
import java.util.Map;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.apache.kafka.common.serialization.StringSerializer;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.config
    .ConcurrentKafkaListenerContainerFactory;
import org.springframework.kafka.core.ConsumerFactory;
import org.springframework.kafka.core
    .DefaultKafkaConsumerFactory;
import org.springframework.kafka.core
    .DefaultKafkaProducerFactory;
```

```
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.kafka.core.ProducerFactory;
import org.springframework.kafka.support.serializer
    .JsonDeserializer;
import org.springframework.kafka.support.serializer
    .JsonSerializer;

import com.santana.dto.ShopDTO;

@Configuration
public class KafkaConfig {

    @Value(value = "${kafka.bootstrapAddress:localhost:9092}")
    private String bootstrapAddress;

    public ProducerFactory<String, ShopDTO> producerFactory() {
        Map<String, Object> props = new HashMap<>();
        props.put(
            ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
            bootstrapAddress);
        props.put(
            ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            StringSerializer.class);
        props.put(
            ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            JsonSerializer.class);
        props.put(
            ProducerConfig.CLIENT_ID_CONFIG,
            "shop-api");
        return new DefaultKafkaProducerFactory<>(props);
    }

    @Bean
    public KafkaTemplate<String, ShopDTO> kafkaTemplate() {
        return new KafkaTemplate<>(producerFactory());
    }

    public ConsumerFactory<String, ShopDTO> consumerFactory() {
        JsonDeserializer<ShopDTO> deserializer =
```

```
            new JsonDeserializer<>(ShopDTO.class);

        Map<String, Object> props = new HashMap<>();
        props.put(
            ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
            bootstrapAddress);

        return new DefaultKafkaConsumerFactory<>(
            props,
            new StringDeserializer(),
            deserializer);
    }

    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, ShopDTO>
        kafkaListenerContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, ShopDTO>
            factory =
            new ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(consumerFactory());
        return factory;
    }

}
```

Fique atento aos imports dos `Serializers` na classe `KafkaConifg`. Devemos utilizar as classes `org.apache.kafka.common.serialization.StringDeserializer` e `org.apache.kafka.common.serialization.StringSerializer`. Existem outras classes com os nomes `StringSerializer` e `JsonSerializer` disponíveis em outras bibliotecas, como a `com.fasterxml.jackson.databind.ser.std.StringSerializer` e, caso o import seja feito para as classes erradas, a aplicação compilará, mas será retornado um erro na hora de salvar os dados no Kafka.

Além disso, o Kafka possui centenas de propriedades que podem melhorar o desempenho da sua aplicação, veremos diversas configurações no decorrer do livro, mas quem quiser já olhar as

configurações possíveis do Kafka pode acessar este endereço da documentação oficial da ferramenta: https://kafka.apache.org/documentation/#configuration.

Com o bean `kafkaTemplate` definido, para enviar a mensagem, apenas temos que criar um método, no caso o `sendMessage`, que receberá um `ShopDTO` e enviará a mensagem para o tópico definido. Note o nome do tópico, o `SHOP_TOPIC`, definido na variável `SHOP_TOPIC_NAME`. Caso esse tópico não exista ainda, ele será criado automaticamente na primeira chamada.

```
package com.santana.events;

import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.stereotype.Service;

import com.santana.dto.ShopDTO;

import lombok.RequiredArgsConstructor;

@Service
@RequiredArgsConstructor
public class KafkaClient {

    private final KafkaTemplate<String, ShopDTO> kafkaTemplate;

    private static final String SHOP_TOPIC_NAME = "SHOP_TOPIC";

    public void sendMessage(ShopDTO msg) {
        kafkaTemplate.send(SHOP_TOPIC_NAME, msg);
    }

}
```

4.3 Enviando os objetos para o tópico

Agora, para enviar as compras criadas para o tópico do Kafka, precisamos fazer uma pequena alteração na classe `ShopController`. Basicamente, precisamos adicionar a relação com a classe `KafkaClient` e também alterar o método `saveShop` para, depois de salvar a compra no banco de dados, enviar a mensagem para o Kafka. A ordem aqui é importante, pois só queremos enviar uma compra para o Kafka caso ela tenha sido salva no banco, então, caso ocorra algum erro na hora de salvar a compra, a mensagem também não será enviada.

```
@RestController
@RequestMapping("/shop")
@RequiredArgsConstructor
public class ShopController {

    private final ShopRepository shopRepository;
    private final KafkaClient kafkaClient;

    // other methods...

    @PostMapping
    public ShopDTO saveShop(@RequestBody ShopDTO shopDTO) {
        shopDTO.setIdentifier(UUID.randomUUID().toString());
        shopDTO.setDateShop(LocalTime.now());
        shopDTO.setStatus("PENDING");

        Shop shop = Shop.convert(shopDTO);
        for (ShopItem shopItem : shop.getItems()) {
            shopItem.setShop(shop);
        }

        shopDTO = ShopDTO.convert(shopRepository.save(shop));
        kafkaClient.sendMessage(shopDTO);
        return shopDTO;

    }

}
```

4.4 Verificando o tópico do Kafka

Executando a aplicação e salvando uma nova compra na rota `POST http://localhost:8080/shop`, o objeto deve ter sido enviado para o Kafka. A listagem a seguir mostra o JSON que eu utilizei para enviar esse objeto para a rota.

```
{
    "items": [
        {
            "productIdentifier": "123456789",
            "amount": 100,
            "price": "1000"
        },
        {
            "productIdentifier": "123456789",
            "amount": 100,
            "price": "1000"
        }
    ]

}
```

E a resposta que foi enviada para o servidor:

```
{
    "identifier": "d351ea15-345d-4068-8f8d-e48812c8169b",
    "dateShop": "10:07:04.848385",
    "items": [
        {
            "productIdentifier": "123456789",
            "amount": 100,
            "price": 1000.0
        },
        {
            "productIdentifier": "123456789",
            "amount": 100,
            "price": 1000.0
        }
```

```
    ]
}
```

Agora, utilizando o comando `bin/kafka-console-consumer.sh --topic SHOP_TOPIC --bootstrap-server localhost:9092 --from-beginning`, podemos verificar que a mensagem já está no Kafka. Note que ela é igual à mensagem que foi retornada como resposta na rota de salvar compras apresentada no capítulo anterior.

```
./bin/kafka-console-consumer.sh \
    --topic SHOP_TOPIC \
    --bootstrap-server localhost:9092 \
    --from-beginning
```

A resposta desse comando devem ser as informações das compras que foram adicionadas no tópico `SHOP_TOPIC`.

```
{"identifier":"d351ea15-345d-4068-8f8d-e48812c8169b","dateShop":
[10,7,4,848385000],"items":
[{"productIdentifier":"123456789","amount":100,"price":1000.0},
{"productIdentifier":"123456789","amount":100,"price":1000.0}]}
```

Se salvarmos outras compras na rota, elas serão adicionadas ao tópico sempre na ordem em que o cadastro da compra foi efetuado.

Agora conseguimos produzir a mensagem e salvar no Kafka. No próximo capítulo, vamos criar um novo microsserviço que será responsável apenas por verificar se os produtos enviados estão disponíveis no estoque e, caso estejam, atualizará a quantidade no estoque e confirmará a compra. Caso não haja disponibilidade, ou um produto inválido seja mandado, esse serviço cancelará a compra.

Capítulo 5
Consumidor

Agora a aplicação já envia os dados da compra para uma fila no Kafka. O próximo passo é verificar se a compra efetuada é válida. Para isso, consumiremos os dados da compra do Kafka, verificaremos se os produtos existem e se o estoque é suficiente para confirmar a compra. Caso as verificações sejam feitas com sucesso, retornaremos uma outra mensagem em outra fila do Kafka indicando que a compra foi confirmada, caso contrário, retornaremos uma mensagem indicando que a compra foi cancelada.

A aplicação que vamos implementar neste capítulo será ao mesmo tempo um consumidor e um produtor, pois ela consumirá as informações da compra de um tópico e salvará as informações do processamento da compra em outro tópico. O nome dessa aplicação é `shop-validator`, já que ela será responsável por validar se a compra é válida ou não.

5.1 Configuração e banco de dados

Para o desenvolvimento do consumidor, vamos criar uma aplicação separada e também utilizaremos o Maven para configurá-la. No arquivo `pom.xml`, teremos que adicionar o cliente do Kafka, como foi feito no projeto anterior, e também adicionar as dependências do `spring-data`, pois neste projeto vamos armazenar no banco de dados as informações dos produtos que estão disponíveis. A listagem a seguir mostra a configuração do projeto.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    https://maven.apache.org/xsd/maven-4.0.0.xsd">
```

```
<modelVersion>4.0.0</modelVersion>

<groupId>com.santana</groupId>
<artifactId>shop-validator</artifactId>
<version>0.0.1-SNAPSHOT</version>

<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.3.5</version>
</parent>

<properties>
    <maven.compiler.source>21</maven.compiler.source>
    <maven.compiler.target>21</maven.compiler.target>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework.kafka</groupId>
        <artifactId>spring-kafka</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
```

```
            <version>1.18.34</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>

</project>
```

O banco de dados dessa aplicação é bastante simples, apenas teremos uma tabela indicando os produtos que existem na aplicação, e qual a quantidade disponível deste produto no estoque. Também vamos inserir alguns produtos no banco de dados, que serão os produtos que teremos disponíveis para serem comprados.

A listagem a seguir mostra o código que deverá ser criado no arquivo `schema.sql` e que contém a criação da tabela `product`.

```
create table product (
    id bigint generated by default as identity,
    identifier varchar(100) not null,
    amount int not null,
    primary key (id)
);
```

A tabela `product` possui três colunas: o `id`, que é um identificador gerado pelo banco de dados; o `identifier`, que é um identificador do produto que utilizaremos na nossa aplicação (esse identificador é o que o usuário deve enviar quando for criar uma nova compra); e a `amount`, que é a quantidade de itens disponíveis para um produto.

Vamos já inserir alguns produtos no banco de dados. Isso deve ser feito no arquivo `data.sql`.

```
insert into product values(1, '123456789', 100);
insert into product values(2, '987654321', 200);
```

Uma nota importante: os arquivos de banco de dados devem ter exatamente os nomes `schema.sql` e `data.sql`, pois o Spring Boot na inicialização da aplicação vai buscar por esses arquivos e, se o nome estiver diferente, os scripts não serão executados.

Outra observação importante, o arquivo `schema.sql` deve ter apenas comandos `DDL` (Data Definition Language), que são os arquivos de definição da base de dados, como `CREATE` e `DROP`, e o arquivo `data.sql` deve ter apenas comandos `DML` (Data Manipulation Language), como o `INSERT` e `UPDATE`.

O último passo na configuração da aplicação é a criação do arquivo `application.properties`, que possuirá exatamente as mesmas configurações do arquivo que desenvolvemos no capítulo 3, a diferença é a porta em que executaremos a aplicação, que agora será a 8081. Faremos isso porque, se as duas aplicações forem executadas na mesma porta, elas não poderão ser executadas no mesmo computador.

```
server.port=8081
spring.jpa.hibernate.ddl-auto=none
```

Os três arquivos `schema.sql`, `data.sql` e `aplication.properties` devem ser colocados na pasta `src/main/resources` do projeto.

5.2 Implementando o consumidor

Com o projeto criado e configurado, vamos iniciar a implementação do consumidor. Novamente será necessário criar a classe com o método `main`, o que é obrigatório em todo projeto do Spring Boot.

```
package com.santana;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure
    .SpringBootApplication;

@SpringBootApplication
public class Main {
```

```
    public static void main(String [] args) {
        SpringApplication.run(Main.class, args);
    }

}
```

Aqui também vamos precisar das classes `ShopItemDTO` e `ShopDTO`. Elas possuem exatamente os mesmos atributos das classes que desenvolvemos no capítulo anterior. Veremos que o Spring Boot criará os objetos com os dados que vieram do Kafka de forma transparente para nós, isto é, uma cópia do objeto que foi enviado para o tópico do Kafka no produtor será criada no consumidor, sem que tenhamos que fazer nada. Apenas colocaremos na classe do consumidor (que criaremos daqui a pouco) um parâmetro usando os DTOs, e o Spring Boot criará o objeto com os dados que vieram do Kafka. Veja a seguir a criação da classe `ShopItemDTO`.

```
package com.santana.dto;

import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
public class ShopItemDTO {
    private String productIdentifier;
    private Integer amount;
    private Float price;
}
```

E agora a criação da classe `ShopDTO`.

```
package com.santana.dto;

import lombok.Getter;
import lombok.Setter;

import java.time.LocalTime;
```

```
import java.util.List;

@Getter
@Setter
public class ShopDTO {
    private String identifier;
    private LocalDate dateShop;
    private String status;
    private List<ShopItemDTO> items = new ArrayList<>();
}
```

Neste projeto, não precisaremos do método `convert`, pois não teremos classes do modelo para esses DTOs, então a implementação aqui é um pouco mais simples.

Assim como no projeto anterior, aqui também teremos que adicionar uma classe para configurar a conexão ao Kafka, a `KafkaConfig`. Não colocarei o código aqui, pois ele é exatamente o mesmo que foi mostrado no capítulo anterior. Você pode copiá-lo e colocar no novo projeto.

> Uma opção interessante seria criar uma biblioteca para as aplicações, com todos os códigos iguais. Ela poderia, por exemplo, conter os DTOs e a configuração do Kafka, assim evitaríamos a duplicação de código.

Uma parte importante dessa aplicação será o armazenamento dos produtos que estão disponíveis. Para isso, precisaremos criar a classe do modelo que representa os dados que virão da tabela `product`, que criamos no começo deste capítulo.

Assim como as classes de modelo do capítulo 3, essa classe possui exatamente os mesmos atributos da tabela, que são o `id`, o `identifier` e o `amount`.

Note também as três anotações, o `@Getter` e o `@Setter` do `lombok` e a `@Entity` para indicar que a classe é uma entidade do banco de

dados.

```
package com.santana.model;

import lombok.Getter;
import lombok.Setter;

import jakarta.persistence.*;

@Getter
@Setter
@Entity(name = "product")
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "identifier")
    private String identifier;

    private Integer amount;

}
```

Também precisaremos do repositório, que é a interface que disponibilizará os métodos para o acesso ao banco de dados. Vimos no capítulo 3 que a interface já disponibiliza diversos métodos, como o `findAll` e o `findById`, mas nesta aplicação precisaremos buscar os produtos pela propriedade `identifier` da classe `Product`.

Não existe um método na interface para buscar pela propriedade `identifier`, por isso temos que criá-lo. Isso é bastante simples, o spring-data possibilita a definição de métodos de busca utilizando apenas um padrão no nome do método. O padrão que utilizaremos é o `findBy`, então basta criar um método na interface com o nome `findBy` + `nome da propriedade`. No nosso caso, o método será `findByIdentifier`, que retornará um produto específico.

Poderíamos também criar o método `findByAmount`, já que `amount` é a outra propriedade que temos na classe `Product`. A única diferença, nesse caso, é que o retorno seria uma lista, já que podemos ter vários produtos que têm o mesmo valor para a propriedade `amount`.

```
package com.santana.repository;

import com.santana.model.Product;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface ProductRepository
    extends JpaRepository<Product, Long> {

    Product findByIdentifier(String identifier);

}
```

Finalmente chegamos à classe principal da aplicação, que será a classe que consumirá as mensagens do tópico do Kafka, a `ReceiveKafkaMessage`. Essa classe possui dois atributos estáticos, que são os nomes dos tópicos que ela acessa: a variável `SHOP_TOPIC_NAME` possui o nome do tópico que receberá a mensagem sobre uma compra, e a `SHOP_TOPIC_EVENT_NAME` possui o nome do tópico que indicará se a compra foi efetuada com sucesso ou não.

A classe possui também os atributos `productRepository`, que é a referência para a interface que fará o acesso ao banco de dados da aplicação, e a `kafkaTemplate`, que é o mesmo objeto que utilizamos no capítulo anterior para enviar mensagens para o Kafka.

A grande novidade agora é o método `listenShopTopic`, que será executado quando mensagens forem adicionados ao tópico `SHOP_TOPIC`. Isso acontecerá por causa da anotação `@KafkaListener` do método. Note que essa anotação possui dois atributos: o primeiro é o nome do tópico que estamos acessando e o segundo é um grupo do qual esse consumidor faz parte, todo consumidor do Kafka deve

fazer parte de um grupo. O conceito de grupos será melhor explicado nos capítulos 7 e 8.

Note ainda que esse método possui um parâmetro do tipo `ShopDTO` e, como dito anteriormente, o Spring Boot vai copiar os dados que estão no Kafka nesse parâmetro do método, transparentemente para o programador. Não será necessário fazer nenhum tipo de mapeamento do JSON que foi adicionado no Kafka para o objeto.

A implementação do método é bastante simples, basicamente o método verifica se todos os produtos da compra existem e se eles estão disponíveis na quantidade que foi solicitada. Caso algum dos produtos não esteja disponível ou não exista, é enviada uma mensagem de volta ao tópico `SHOP_TOPIC_EVENT` com o status `ERROR`. Caso todos os produtos estejam disponíveis, é retornada uma mensagem com o status `SUCCESS`.

Note que estamos enviando o mesmo objeto de volta no tópico, mas isso não é obrigatório. Poderíamos criar um objeto DTO diferente com apenas o identificador da compra e o status, por exemplo, mas para simplificar a implementação vou utilizar sempre o mesmo objeto.

```
package com.santana.events;

import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.stereotype.Service;

import com.santana.dto.ShopDTO;
import com.santana.dto.ShopItemDTO;
import com.santana.model.Product;
import com.santana.repository.ProductRepository;

import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;

@Slf4j
```

```java
@Service
@RequiredArgsConstructor
public class ReceiveKafkaMessage {

    private static final String
        SHOP_TOPIC_NAME = "SHOP_TOPIC";

    private static final String
        SHOP_TOPIC_EVENT_NAME = "SHOP_TOPIC_EVENT";

    private final ProductRepository productRepository;

    private final KafkaTemplate<String, ShopDTO> kafkaTemplate;

    @KafkaListener(topics = SHOP_TOPIC_NAME, groupId = "group")
    public void listenShopTopic(ShopDTO shopDTO) {
        try {
            log.info("Compra recebida no tópico: {}.",
                shopDTO.getIdentifier());

            boolean success = true;
            for (ShopItemDTO item : shopDTO.getItems()) {
                Product product = productRepository
                        .findByIdentifier(
                            item.getProductIdentifier());

                if (!isValidShop(item, product)) {
                    shopError(shopDTO);
                    success = false;
                    break;
                }

            }

            if (success) {
                shopSuccess(shopDTO);
            }
        } catch(Exception e) {
            log.error("Erro no processamento da compra {}",
```

```
                shopDTO.getIdentifier());
        }

    }

    // valida se a compra possui algum erro
    private boolean isValidShop(
        ShopItemDTO item,
        Product product) {

        return product != null &&
            product.getAmount() >= item.getAmount();
    }

    // Envia uma mensagem para o Kafka indicando erro na compra
    private void shopError(ShopDTO shopDTO) {
        log.info("Erro no processamento da compra {}.",
            shopDTO.getIdentifier());
        shopDTO.setStatus("ERROR");
        kafkaTemplate.send(SHOP_TOPIC_EVENT_NAME, shopDTO);
    }

    // Envia uma mensagem para o Kafka indicando sucesso na compra
    private void shopSuccess(ShopDTO shopDTO) {
        log.info("Compra {} efetuada com sucesso.",
            shopDTO.getIdentifier());
        shopDTO.setStatus("SUCCESS");
        kafkaTemplate.send(SHOP_TOPIC_EVENT_NAME, shopDTO);
    }

}
```

Note que o consumidor está dentro de um `try catch`. É bom garantir que o método do consumidor não vai retornar erros, do contrário, serão feitas várias tentativas de reprocessar a mensagem. Veremos com mais detalhes como funcionam os reprocessamentos no capítulo 10.

Uma outra novidade interessante é a anotação `@Slf4j` do `lombok`. Essa anotação cria um objeto `log`, que poderemos utilizar para criar logs em nossa aplicação. Note que logo na primeira linha do método estou criando uma mensagem indicando qual o identificador da compra que foi recebida, além dos logs que indicam se a compra foi efetuada com sucesso ou se houve algum erro no processamento da compra.

5.3 Execução da aplicação

Para executar a aplicação, é preciso inicializar o Kafka e também rodar os produtos. Depois disso, inicialize o consumidor. Como não temos uma interface direta para essa aplicação, ela apenas receberá e enviará mensagens para o Kafka. Vamos verificar os logs da aplicação e verificar se foram geradas mensagens no tópico `SHOP_TOPIC_EVENT`. Primeiro, faremos uma compra válida. Para isso, eu enviarei o JSON a seguir para a rota `POST http://localhost:8080/shop`.

```
{
    "items": [
        {
            "productIdentifier": "123456789",
            "amount": 1,
            "price": "1000"
        }
    ]

}
```

Note que a compra é válida, pois incluímos este produto no banco de dados no arquivo `data.sql` no início do capítulo. Se a aplicação

funcionar corretamente, veremos no log do consumidor as seguintes linhas:

```
2021-07-24 12:33:50.695  INFO 52877 --- [ntainer#0-0-C-1]
com.santana.events.ReceiveKafkaMessage   : Compra recebida no tópico:
17edb4a6-a8a6-4c34-b2e4-45514696b24d.
2021-07-24 12:33:50.828  INFO 52877 --- [ntainer#0-0-C-1]
com.santana.events.ReceiveKafkaMessage   : Compra 17edb4a6-a8a6-4c34-b2e4-
45514696b24d efetuada com sucesso.
```

Além disso, podemos iniciar um consumidor no tópico `SHOP_TOPIC_EVENT` para verificar se a compra foi efetuada com sucesso também.

```
./bin/kafka-console-consumer.sh \
    --topic SHOP_TOPIC_EVENT \
    --bootstrap-server localhost:9092
```

Se tudo funcionou corretamente, veremos a compra com o status `SUCCESS` no tópico.

```
{"identifier":"17edb4a6-a8a6-4c34-b2e4-45514696b24d","dateShop":
[2021,7,24],"status":"SUCCESS","items":
[{"productIdentifier":"123456789","amount":1,"price":1000.0}]}
```

Na sequência, podemos fazer o mesmo para uma compra inválida, por exemplo, utilizando o seguinte JSON:

```
{
    "items": [
        {
            "productIdentifier": "123",
            "amount": 1,
            "price": "1000"
        }
    ]

}
```

Essa compra é inválida, pois o produto com identificador 123 não existe no banco de dados da `shop-validator`, então o método `isValidShop` retornará `false`, e a compra ficará com o status `ERROR`. Veremos então no log as seguintes linhas:

```
2021-07-24 12:34:23.158  INFO 52877 --- [ntainer#0-0-C-1]
com.santana.events.ReceiveKafkaMessage   : Compra recebida no tópico:
9d0bddac-b3eb-40e4-a8ce-ac1584a0d91b.
2021-07-24 12:34:23.160  INFO 52877 --- [ntainer#0-0-C-1]
com.santana.events.ReceiveKafkaMessage   : Erro no processamento da compra
9d0bddac-b3eb-40e4-a8ce-ac1584a0d91b.
```

Utilizando o mesmo consumidor que iniciamos para ver a compra com sucesso, também veremos as seguintes informações para essa nova compra. Mote o status `ERROR`:

```
{"identifier":"9d0bddac-b3eb-40e4-a8ce-ac1584a0d91b","dateShop":
[2021,7,24],"status":"ERROR","items":
[{"productIdentifier":"123","amount":1,"price":1000.0}]}
```

Agora o processamento da compra está sendo efetuado com sucesso, temos apenas que finalizar o ciclo e atualizar o status da compra na primeira aplicação, para que a rota `GET http://localhost:8080/shop/{shopIdentifier}` retorne a compra com o status atualizado. Vamos fazer isso no próximo capítulo.

Capítulo 6
Finalizando a shop-api

Nossa aplicação já produz e consome mensagens da fila do Kafka. Neste processo estamos recebendo as compras e depois verificando se elas podem ser concluídas com sucesso ou não. Porém, ainda não estamos disponibilizando o status final da compra para o usuário, pois, para isso ser feito, precisaremos que a aplicação desenvolvida nos capítulos 3 e 4 receba a mensagem do tópico `SHOP_TOPIC_EVENT` que acabamos de criar no capítulo anterior. Recebendo essa mensagem, a aplicação atualizará o status da compra, que estará `PENDING`, para `SUCCESS` se ela tiver sido efetuada com sucesso ou para `ERROR` caso algum produto não exista ou se a quantidade de algum produto for insuficiente.

Precisaremos de apenas duas alterações na `shop-api` desenvolvida no capítulo 3 e 4. Primeiro, devemos alterar a interface `ShopRepository` para incluir o método `findByIdentifier`, que fará a busca de um objeto do tipo `Shop` pelo seu identificador. O código a seguir mostra a interface `ShopRepository` completa.

```
package com.santana.repository;

import com.santana.model.Shop;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface ShopRepository
    extends JpaRepository<Shop, Long> {

    public Shop findByIdentifier(String identifier);

}
```

A outra mudança é que devemos criar um consumidor da fila `SHOP_TOPIC_EVENT`, que receberá o status da compra, `SUCCESS` ou `ERROR`. O consumidor está implementado no método `listenShopEvents`. Esse consumidor receberá os dados da fila, fará a busca da compra no banco de dados e apenas atualizará o status no banco de dados. O código a seguir mostra a implementação dessa classe.

```
package com.santana.events;

import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Service;

import com.santana.dto.ShopDTO;
import com.santana.model.Shop;
import com.santana.repository.ShopRepository;

import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;

@Slf4j
@Service
@RequiredArgsConstructor
public class ReceiveKafkaMessage {

    private final ShopRepository shopRepository;

    private static final String SHOP_TOPIC_EVENT_NAME
        = "SHOP_TOPIC_EVENT";

    @KafkaListener(
        topics = SHOP_TOPIC_EVENT_NAME,
        groupId = "group")
    public void listenShopEvents(ShopDTO shopDTO) {
        try {
            log.info("Status da compra recebida no tópico: {}."
                , shopDTO.getIdentifier());

            Shop shop = shopRepository
```

```
                .findByIdentifier(shopDTO.getIdentifier());
            shop.setStatus(shopDTO.getStatus());
            shopRepository.save(shop);
        } catch(Exception e) {
            log.error("Erro no processamento da compra {}"
                , shopDTO.getIdentifier());
        }
    }

}
```

6.1 Execução da aplicação

Pronto, agora nosso sistema está completo. A primeira aplicação recebe as informações da compra e envia essas informações para a fila, a segunda aplicação recebe as informações da fila e, com essas informações, verifica se a compra foi efetuada com sucesso. Depois, ela envia outra mensagem para o Kafka indicando o status final da compra. Faremos dois testes agora chamando as rotas que o usuário pode chamar, a `POST /shop` e a `GET /shop`. Primeiro vamos fazer uma execução em que a compra será efetuada com sucesso.

Para isso, devemos fazer uma chamada para a rota `POST /shop` enviando quais itens serão comprados. Todos os itens devem existir e possuir quantidade suficiente para que a compra seja efetuada com sucesso. O `json` a seguir é um exemplo de uma compra que será efetuada com sucesso, pois o produto passado no campo `productIdentifier` existe no banco de dados, e existe estoque suficiente para aceitar a quantidade passada no campo `amount`.

```
{
    "items": [
        {
            "productIdentifier": "123456789",
            "amount": "1",
```

```
            "price": "1000"
        }
    ]
}
```

A resposta dessa chamada é mostrada no `json` a seguir, que tem dois campos importantes: o `identifier`, que é o id da compra, para que possamos verificar o status da compra depois; e o `status`, que indica que a compra ainda está sendo processada, pois ela está no status `PENDING`.

```
{
    "identifier": "08818317-9e28-4392-ac65-2c8564483da4",
    "dateShop": "2021-08-26",
    "status": "PENDING",
    "items": [
        {
            "productIdentifier": "123456789",
            "amount": 1,
            "price": 1000.0
        }
    ]
}
```

Depois de alguns segundos, podemos fazer uma chamada para a rota `GET /shop`, que trará os detalhes da compra efetuada anteriormente. Note que a resposta dessa chamada é parecida com a resposta da chamada `POST /shop`, a diferença é que agora a compra já foi validada no `shop-validator`, então o status foi alterado para `SUCCESS`.

```
[
    {
        "identifier": "08818317-9e28-4392-ac65-2c8564483da4",
        "dateShop": "2021-08-26",
        "status": "SUCCESS",
        "items": [
            {
                "productIdentifier": "123456789",
```

```
                "amount": 1,
                "price": 1000.0
            }
        ]
    }
]
```

Agora vamos fazer um segundo teste enviando um produto inválido para a realização da compra. Para isso, podemos fazer uma segunda chamada para a rota `POST /shop`, mas agora, o `productIdentifier` enviado não existe no banco de dados, por isso a compra deve ser rejeitada.

```
{
    "items": [
        {
            "productIdentifier": "123",
            "amount": "1",
            "price": "1000"
        }
    ]
}
```

Porém, note que a resposta da chamada também retornará o `status` como `PENDING`, pois a compra ainda não foi processada no momento em que a resposta é gerada.

```
{
    "identifier": "66f878de-2571-4ece-876c-f1cd84b68c36",
    "dateShop": "2021-08-26",
    "status": "PENDING",
    "items": [
        {
            "productIdentifier": "123",
            "amount": 1,
            "price": 1000.0
        }
    ]
}
```

Depois de alguns segundos, podemos fazer uma chamada para a rota `GET /shop`. Nela, veremos todas as compras que foram feitas na aplicação. Podemos verificar agora o campo `status` da última compra, que está com o status `ERROR`, pois o `productIdentifier` era inválido.


```
[
    {
        "identifier": "66f878de-2571-4ece-876c-f1cd84b68c36",
        "dateShop": "2021-08-26",
        "status": "ERROR",
        "items": [
            {
                "productIdentifier": "123",
                "amount": 1,
                "price": 1000.0
            }
        ]
    },
    {
        "identifier": "08818317-9e28-4392-ac65-2c8564483da4",
        "dateShop": "2021-08-26",
        "status": "SUCCESS",
        "items": [
            {
                "productIdentifier": "123456789",
                "amount": 1,
                "price": 1000.0
            }
        ]
    }
]
```


Terminamos a implementação da parte principal da aplicação, que é composta por duas partes: a api que recebe as compras, e o serviço que faz a validação das compras. Como estamos utilizando o Kafka, podemos implementar outras funcionalidades na aplicação sem que tenhamos que alterar nada no código desenvolvido até agora, isso porque o Kafka permite que mais de um consumidor se conecte a

um tópico, tanto para fazer processamentos diferentes quanto para paralelizar um mesmo processamento.

Essas duas funcionalidades serão vistas nos próximos dois capítulos. No próximo, criaremos uma nova aplicação que gerará relatórios com a quantidade de vendas feitas na aplicação e veremos que para fazer isso, não precisaremos alterar nada no código desenvolvido até aqui.

CAPÍTULO 7
Diferentes grupos de consumidores

Imagine que queremos ter diferentes aplicações conectando em uma fila do Kafka, por exemplo. Na nossa aplicação, já temos duas filas, uma para enviar as compras que precisam ser confirmadas e outra que retorna o resultado da compra, se ela foi aceita ou não. Podemos agora querer desenvolver uma terceira aplicação para gerar relatórios. Ela vai sempre salvar o identificador da compra e o status, que indica se a compra foi realizada com sucesso ou não. Podemos fazer isso sem ter que alterar nada do que já foi desenvolvido, pois o Kafka permite a criação de diferentes `Consumer Groups`, e cada um desses grupos pode ler todas as mensagens de um mesmo tópico independentemente do que é feito em outros grupos.

Neste capítulo, vamos ver como o Kafka faz a distribuição das mensagens para os diferentes grupos de consumidores. Também vamos implementar mais um serviço que vai se conectar ao tópico que envia os status das compras no Kafka e contabilizar o número de compras que foram confirmadas ou não, permitindo que o usuário verifique os números de compras em uma rota REST.

7.1 Consumer Groups

`Consumer Group` é a ferramenta utilizada pelo Kafka para balancear e distribuir as mensagens de um tópico entre diferentes consumidores. Um consumidor do Kafka deve sempre participar de um grupo, e um tópico do Kafka pode ser acessado por diferentes grupos de consumidores. Isso definirá como o Kafka vai enviar as mensagens para os consumidores.

Se existirem dois ou mais consumidores em um mesmo grupo, o Kafka vai enviar as mensagens para apenas um dos consumidores dentro do grupo, isto é, a mensagem de um tópico será lida por apenas um consumidor de cada grupo. O Kafka tenta sempre balancear o número de mensagens enviadas para consumidores de um mesmo grupo. Normalmente, consumidores de um mesmo grupo são diversas instâncias de uma mesma aplicação, que foram criadas para acelerar o processamento das mensagens.

Por exemplo, imagine que temos um tópico de compras que foram processadas, e devemos enviar um e-mail para o comprador informando que a compra foi realizada com sucesso. Podemos ter um consumidor que lê esse tópico e envia a mensagem para o comprador. Porém, em épocas como o Natal ou a Black Friday, o número de compras pode crescer muito, e apenas um consumidor pode não dar conta de todo o processamento, fazendo com que o tempo para o envio do e-mail fique muito grande. Podemos escalar essa aplicação para que sejam criadas diversas instâncias dessa aplicação e, se todos os consumidores estiverem no mesmo grupo, o Kafka fará a distribuição das mensagens entre os consumidores automaticamente.

Já consumidores de diferentes grupos leem as mensagens independentemente dos outros grupos, isto é, as mesmas mensagens serão enviadas para todos os grupos. Normalmente utilizamos grupos diferentes para aplicações diferentes. Por exemplo, se além da aplicação do e-mail, temos uma outra que precisa ler as compras que foram processadas com sucesso para a geração de um relatório, basta criar uma outra aplicação, com outro consumidor que pertence a outro grupo, que ela também receberá todas as mensagens enviadas para o tópico.

Resumindo, todos os grupos sempre recebem todas as mensagens enviadas para o tópico. Cada grupo funciona independentemente dos outros grupos e as mensagens enviadas para um grupo não influenciam nas mensagens enviadas para outro grupo. Já dentro dos grupos, não existe duplicação no consumo das mensagens, cada

mensagem é lida por apenas um consumidor daquele grupo. Esses controles são feitos internamente dentro do Kafka, sem que tenhamos que implementar nada na nossa aplicação.

A figura a seguir mostra essa ideia. Inicialmente, dois produtores enviam seis mensagens para o topico-1 do Kafka. É possível observar que existem dois grupos consumindo as mensagens desse tópico, o cg-1 e o cg-2. Note que todas as mensagens são enviadas tanto para o cg-1 quanto para o cg-2. Note também que o cg-1 possui três consumidores (c-1, c-2 e c-3), e o cg-2 possui apenas um consumidor (c-4). As mensagens são divididas no cg-1, onde cada consumidor recebe apenas duas mensagens do tópico; já no cg-2, o único consumidor recebe todas as mensagens que existem no tópico.

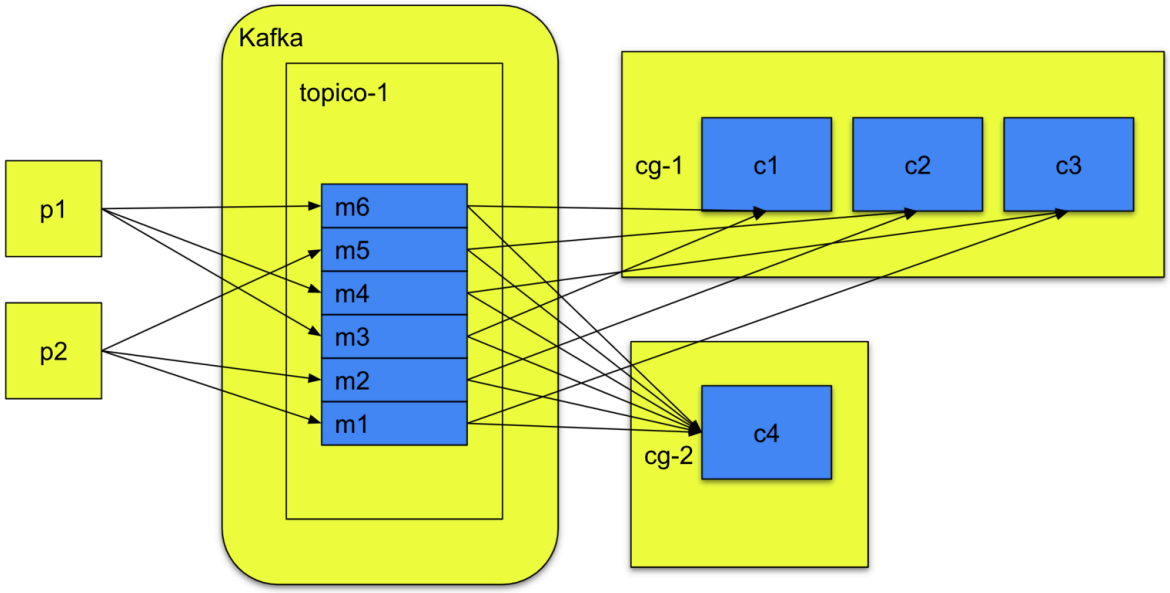


Figura 7.1: Funcionamento do Kafka.

Note que os diferentes grupos recebem as mesmas mensagens. Já nos grupos, cada consumidor (caso exista mais de um) recebe mensagens diferentes. Para saber quais as mensagens que cada `Consumer Group` já recebeu, o Kafka mantém um índice de qual mensagem foi a última recebida pelo grupo. Diferentemente de

outras ferramentas de mensageria, o Kafka mantém os dados no tópico por um período configurável, que por padrão é 7 dias. Assim, os dados podem ser lidos mesmo depois de terem sido consumidos por outros grupos.

Outro conceito bastante importante relacionado aos `Consumer Group` são as **partições**. Um tópico do Kafka pode ser composto por uma ou mais partições, que são divisões dentro de um tópico. Quando uma mensagem chega a um tópico, ela será alocada em uma partição do tópico, e apenas um consumidor de um grupo pode estar conectado a uma partição por vez. É com esse mecanismo que o Kafka evita que mais de um consumidor de um grupo leia a mesma mensagem. As partições serão melhor explicadas no próximo capítulo, onde também mostraremos a implementação de uma aplicação que faz uso das partições.

7.2 Implementando a nova aplicação

Para mostrar a ideia dos `Consumer Groups`, vamos criar uma nova aplicação que vai se conectar também ao tópico `SHOP_TOPIC_EVENT`. Ele fará a leitura das mensagens e totalizará o número de compras que foram efetuadas com sucesso e o número de compras que tiveram erro. Esses dados serão disponibilizados em uma rota REST que também será criada nesta aplicação.

O nome dessa aplicação será `shop-report`. Ela será independente das aplicações implementadas nos capítulos anteriores, assim o consumidor dessa aplicação estará em um `Consumer Group` diferente, para que ela acesse todas as mensagens enviadas para o Kafka.

Configuração da aplicação

Temos que fazer as mesmas configurações que fizemos nas aplicações anteriores para criar a nova aplicação. O primeiro passo é

a criação do arquivo `pom.xml`, que é praticamente igual ao das aplicações `shop-api` e `shop-validator`. As dependências são as mesmas, com a única diferença no nome da nova aplicação, que foi definido na tag `artifactId`.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.3.5</version>
    </parent>

    <groupId>org.example</groupId>
    <artifactId>shop-report</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>21</maven.compiler.source>
        <maven.compiler.target>21</maven.compiler.target>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.kafka</groupId>
            <artifactId>spring-kafka</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
```

```
        </dependency>
        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.34</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
</project>
```

Também precisaremos criar o arquivo `application.properties`. A única diferença entre esse arquivo e as outras aplicações é a porta, definida na propriedade `server.port`. Para essa aplicação, utilizaremos a porta 8082.

```
server.port=8082
spring.jpa.hibernate.ddl-auto=none
```

Agora, podemos criar as tabelas do banco de dados que armazenará a quantidade de compras realizadas com sucesso e a quantidade de compras que tiveram erro no processamento. Para isso, utilizaremos uma tabela bem simples que terá um id, um identificador do status da compra e a quantidade de compras nesse status. Como só temos dois possíveis status de compras em nossa aplicação, só teremos dois registros nessa tabela, um para o status `SUCCESS` e outro para o status `ERROR`. O script a seguir, que cria a tabela, deve ficar no arquivo `schema.sql`.

```
create table shop_report (
    id bigint generated by default as identity,
    identifier varchar(100) not null,
    amount int not null,
    primary key (id)
);
```

Já vamos também inicializar o banco de dados com os registros para os dois possíveis status das compras, e iniciar os valores com 0, pois inicialmente nenhuma compra terá sido realizada. Esse script deve ficar no arquivo `data.sql`.

```
insert into shop_report(identifier, amount) values('SUCCESS', 0);
insert into shop_report(identifier, amount) values('ERROR', 0);
```

7.3 Implementando o consumidor

Assim como fizemos no capítulo 6, vamos criar um consumidor para o tópico `SHOP_TOPIC_EVENT`. Para isso, vamos precisar de uma das classes que já utilizamos nos capítulos anteriores. Uma delas é a `ShopDTO`, que possui o status da compra, indicando se ela foi realizada com sucesso ou não. Note na listagem a seguir que no código da aplicação `shop-report`, essa classe não possui a data da compra e a lista de itens da compra, isso porque nessa aplicação não vamos utilizar esses dados, então podemos simplesmente ignorar os campos que não vamos utilizar.

```
package com.santana.dto;

import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
public class ShopDTO {
    private String identifier;
    private String status;
}
```

No banco de dados da aplicação `shop-report`, vamos salvar dois registros, um que contabilizará o número de compras realizadas com

sucesso e outro que contabilizará o número de compras com erro. Para isso, precisamos criar a classe do modelo que representa a tabela `shop_report`. Ela possui apenas dois campos, o `identifier`, que terá o status `SUCCESS` ou `ERROR`, e o `amount`, que terá a quantidade de compras em cada status. A listagem a seguir mostra o código da classe `ShopReport`.

```
package com.santana.model;

import lombok.Getter;
import lombok.Setter;

import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;

@Getter
@Setter
@Entity(name = "shop_report")
public class ShopReport {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String identifier;

    private Integer amount;

}
```

Sempre que os dados de uma compra forem recebidos, vamos executar um `update` na tabela `shop_report` incrementando as compras com sucesso ou com erro. Para isso, precisamos criar a interface `ReportRepository`, que executará esse comando no banco de dados. Note que definimos o método `incrementShopStatus`, que possui

a anotação `@Query` com o update que será executado no banco de dados. Além disso, esse método possui a anotação `@Modifying`, que é obrigatória sempre que o método executar um comando `INSERT`, `DELETE` ou `UPDATE`, indicando que haverá alterações nos dados. Caso o comando seja um `SELECT`, a anotação `@Modifying` não será necessária. O código a seguir mostra a implementação dessa interface.

```
package com.santana.repository;

import com.santana.model.ShopReport;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Modifying;
import org.springframework.data.jpa.repository.Query;
import org.springframework.stereotype.Repository;

@Repository
public interface ReportRepository extends JpaRepository<ShopReport, Long>
{

    @Modifying
    @Query(value = "update shop_report "
            + "set amount = amount + 1 "
            + "where identifier = :shopStatus",
            nativeQuery = true)
    void incrementShopStatus(@Param(value = "shopStatus") String
shopStatus);

}
```

Depois podemos implementar a classe que terá o método que consumirá as mensagens do Kafka, que é a classe `ReceiveKafkaMessage`. O método `listenShopTopic` receberá todas as mensagens que serão inseridas no tópico `SHOP_TOPIC_EVENT`. Note, na anotação `@KafkaListener`, o atributo `groupId`. Nessa aplicação, definimos o grupo como `group_report`, que é diferente do grupo `group` definido no capítulo anterior. Os consumidores pertencem a grupos diferentes, logo os consumidores definidos aqui neste

capítulo e o definido no capítulo anterior receberão exatamente as mesmas mensagens.

Isso demonstra o conceito de `Consumer Groups` discutido no início do capítulo: temos dois consumidores para o mesmo tópico do Kafka, mas, como eles pertencem a grupos diferentes, eles receberão todas as mensagens enviadas para o tópico. O código a seguir mostra a implementação do consumidor.

```
package com.santana.events;

import jakarta.transaction.Transactional;

import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Service;

import com.santana.dto.ShopDTO;
import com.santana.repository.ReportRepository;

import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;

@Slf4j
@Service
@RequiredArgsConstructor
public class ReceiveKafkaMessage {

    private static final String SHOP_TOPIC_EVENT_NAME
        = "SHOP_TOPIC_EVENT";

    private final ReportRepository reportRepository;

        @Transactional
        @KafkaListener(
                topics = SHOP_TOPIC_EVENT_NAME,
                groupId = "group_report")
        public void listenShopTopic(ShopDTO shopDTO) {
            try {
            log.info("Compra recebida no tópico: {}.",
```

```
                    shopDTO.getIdentifier());
            reportRepository
                .incrementShopStatus(shopDTO.getStatus());
            } catch (Exception e) {
                log.error(
                    "Erro no processamento da mensagem", e);
            }
        }
}
```

Na implementação do método `listenShopTopic`, apenas recebemos uma informação da compra e chamamos o método do repositório que incrementa as compras com sucesso ou com erro que tivemos em nossa aplicação. Note a anotação `@Transactional` também no método, ela é necessária sempre que o método for fazer alterações nos dados. Então, como estamos fazendo um `update` nos dados da tabela `shop-report`, temos que colocar essa anotação.

7.4 Implementando a rota REST

Agora que o banco de dados já está sendo atualizado com os dados recebidos do Kafka, podemos criar uma rota que apenas retornará esses dados para o usuário. O primeiro passo para isso é criar uma classe que terá os dados que serão retornados na rota, a classe `ShopReportDTO`. Note que ela possui apenas os campos `identifier` e `amount`, além do método `convert`, que converte um objeto do tipo `ShopReport` para um objeto do tipo `ShopReportDTO`. O código a seguir mostra a implementação dessa classe.

```
package com.santana.dto;

import com.santana.model.ShopReport;

import lombok.Getter;
import lombok.Setter;
```

```
@Getter
@Setter
public class ShopReportDTO {

    private String identifier;

    private Integer amount;

    public static ShopReportDTO convert(ShopReport shopReport) {
        ShopReportDTO shopDTO = new ShopReportDTO();
        shopDTO.setIdentifier(shopReport.getIdentifier());
        shopDTO.setAmount(shopReport.getAmount());
        return shopDTO;
    }

}
```

Por fim, apenas implementamos um `@RestController` que disponibilizará uma rota para o usuário recuperar a quantidade de compras realizadas com sucesso e a quantidade de compras com erro. Na anotação `@RequestMapping`, é definido o caminho para essa rota, que é implementada no método `getShopReport`. Esse método apenas chama o repositório para recuperar todos os registros da tabela `shop_report`. O código a seguir mostra a implementação dessa classe.

```
package com.santana.controller;

import com.santana.dto.ShopReportDTO;
import com.santana.repository.ReportRepository;

import lombok.RequiredArgsConstructor;
import org.springframework.web.bind.annotation.*;

import java.util.List;
import java.util.stream.Collectors;

@RestController
@RequestMapping("/shop_report")
```

```
@RequiredArgsConstructor
public class ShopController {

    private final ReportRepository reportRepository;

    @GetMapping
    public List<ShopReportDTO> getShopReport() {
        return reportRepository.findAll()
                .stream()
                .map(shop -> ShopReportDTO.convert(shop))
                .collect(Collectors.toList());
    }

}
```

7.5 Testando a aplicação

Podemos testar a nossa aplicação agora. Se chamarmos a rota `GET /shop_report`, ela mostrará as compras que foram realizadas com sucesso ou não. Caso a chamemos antes de efetuar alguma compra, ela retornará 0 tanto para compras com sucesso quanto para compras com erro, como no exemplo a seguir.

```
[
    {
        "identifier": "SUCCESS",
        "amount": 0
    },
    {
        "identifier": "ERROR",
        "amount": 0
    }
]
```

Porém, se realizarmos algumas compras chamando a rota `POST /shop` da `shop-api`, como já fizemos no capítulo 3 e 6, veremos que o

contador das compras realizadas com sucesso ou erro será alterado, incrementando as compras com sucesso quando o processamento for efetuado com sucesso, e com erro quando algum problema ocorrer. O exemplo a seguir mostra a chamada a rota `GET /shop_report` depois do processamento de 12 compras efetuadas com sucesso e 4 que retornaram erros.

```
[
    {
        "identifier": "SUCCESS",
        "amount": 12
    },
    {
        "identifier": "ERROR",
        "amount": 4
    }
]
```

Com a aplicação desenvolvida neste capítulo, já conseguimos criar diferentes aplicações que se conectam a uma fila do Kafka. Porém, ainda não conseguimos paralelizar o processamento de uma mesma aplicação. Se a quantidade de compras que chegam à nossa aplicação ficar muito grande, pode ser que a fila comece a crescer demais, assim a espera para o processamento ficará muito grande. Para resolver esse problema, precisaremos ter vários consumidores que fazem a mesma tarefa, como processar as compras, mas para isso funcionar, precisaremos explorar melhor as partições do Kafka. No próximo capítulo, descobriremos como fazer isso.

Capítulo 8
Paralelizando tarefas

Existem situações em que o tráfego da aplicação muda bastante. Por exemplo, em uma data especial como Natal, o número de compras em um uma loja pode ter um grande aumento, e se o processo de verificação de compra for demorado, a quantidade de mensagens na fila pode começar a crescer rapidamente, por isso, devemos aumentar o número de consumidores que processam as compras.

Com o Kafka, paralelizar o processamento da compra em vários consumidores é bastante simples com os `Consumer Groups` que vimos no capítulo anterior. Veremos que não será necessário fazer nenhuma mudança no código das aplicações para paralelizar o consumo das mensagens, mas teremos que recriar o nosso tópico, para que ele tenha mais de uma partição. As partições serão o principal assunto deste capítulo, pois são elas que permitem a paralelização de tarefas no Kafka de forma transparente para quem desenvolve.

8.1 Partições

O principal conceito do Kafka são os tópicos, que já utilizamos em nossa aplicação nos últimos capítulos. Cada tópico do Kafka é dividido em uma ou mais partições, e cada consumidor se conecta a uma ou mais partições.

A partição é o principal mecanismo para possibilitar a distribuição das mensagens entre os consumidores de um mesmo grupo, permitindo a paralelização do processamento das mensagens de um

tópico. Ela é usada para garantir que apenas um consumidor de um grupo vai processar uma mensagem específica, sem que haja duplicação no processamento.

As partições funcionam como listas: quando uma mensagem chega ao tópico, ela é adicionada a uma das partições e já sabemos que o Kafka sempre balanceia o número de mensagens pelas partições para que elas tenham mais ou menos a mesma quantidade de mensagens. Sempre que um consumidor se conecta a um tópico, o Kafka mapeia esse consumidor em uma ou mais partições, e o consumidor vai ler mensagens apenas da partição em que ele está mapeado.

A figura a seguir, que estende a figura do capítulo anterior, exemplifica como funcionam as partições no Kafka.

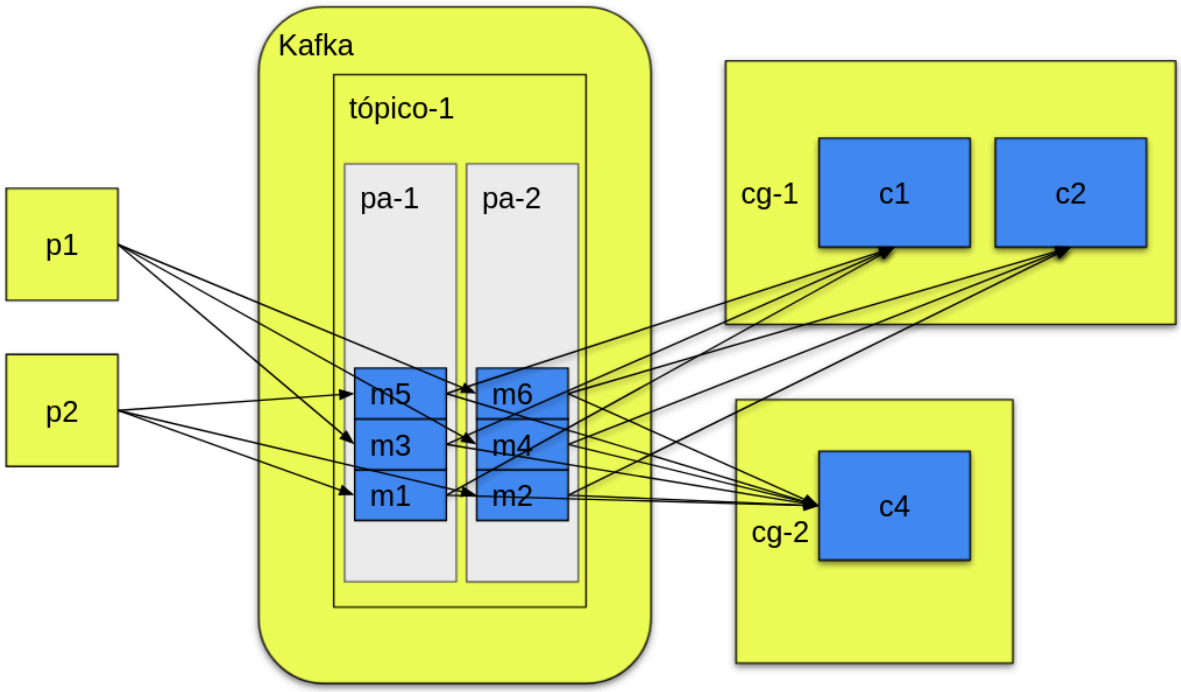


Figura 8.1: Funcionamento do Kafka com partições.

Nessa imagem, note que o tópico é formado por duas partições, a pa-1 e a pa-2, e que todas as mensagens da pa-1 são enviadas para o c-1, e todas as mensagens do pa-2 são enviadas para o c-2. Como

o tópico possui apenas duas partições, não adiantaria ter mais consumidores, que eles ficariam ociosos. Note também que as partições não mudam em nada o funcionamento dos produtores, pois quem decidirá em qual partição uma mensagem será alocada é o Kafka, tornando o processo totalmente transparente para os produtores (no próximo capítulo, veremos o conceito de chaves que muda um pouco isso).

Ainda na figura, como o cg-2 possui apenas um consumidor, nada é alterado nele, o c-4 continua processando todas as mensagens que são colocadas no tópico. Além disso, as partições também definem duas coisas que todo desenvolvedor que utiliza o Kafka deve saber:

1. As mensagens não possuem uma ordem definida para serem consumidas em tópicos com mais de uma partição. Por exemplo, a mensagem-1 chega antes da mensagem-2 em um tópico, mas elas são alocadas em partições diferentes. Se o consumidor que está associado à partição da mensagem-2 estiver mais adiantado que o consumidor que está lendo a partição da mensagem-1, a mensagem-2 será processada antes, mesmo tendo sido inserida antes no tópico. Se a ordem das mensagens for um requisito importante, precisaremos utilizar o conceito de chaves, que veremos no capítulo 9, ou utilizar apenas uma partição.

2. Cada partição terá apenas um consumidor recebendo mensagem, por isso, não devem existir mais consumidores do que partições disponíveis, senão os consumidores a mais ficarão ociosos. O contrário não é problema, se existirem mais partições do que consumidores, o Kafka balanceará o número de partições por consumidores. Por exemplo, se existirem seis partições e três consumidores, cada consumidor vai ler mensagens de duas partições, se existirem cinco partições e três consumidores, dois consumidores vão se conectar a duas partições e um consumidor vai se conectar a uma partição.

Offsets

Todas as mensagens do Kafka dentro de uma partição possuem um `offset`, que nada mais é do que um índice indicando a posição da mensagem na lista de mensagens em uma partição. Esse índice é bastante utilizado pelo Kafka para determinar quais mensagens enviar para um consumidor, pois o Kafka utiliza os `offsets` para saber qual a última mensagem lida por um grupo em uma partição.

Por exemplo, em um caso onde existem 20 mensagens em uma partição, os `offsets` vão de 0 a 19. Um consumidor começa a ler essas mensagens, mas ele processa apenas as 15 primeiras e para de executar. Se outro consumidor começar a executar no mesmo grupo que o primeiro consumidor, não vamos querer que ele leia todas as mensagens novamente, mas sim que ele comece a processar a partir da decima sexta mensagem. O Kafka utiliza os `offsets` para fazer esse tipo de controle.

Rebalanceamento

O Kafka sempre faz o rebalanceamento dos consumidores e das partições automaticamente, sem que tenhamos que fazer nenhuma configuração especial para isso. Se temos um tópico com 3 partições e inicialmente temos apenas um consumidor em um grupo para esse tópico, as 3 partições serão alocadas para esse consumidor. Se iniciarmos um segundo consumidor neste mesmo grupo para o mesmo tópico, o Kafka automaticamente fará o rebalanceamento das partições, alocando 1 partição para um consumidor e 2 para o outro. Se iniciarmos mais um consumidor, a distribuição ficará perfeita, e cada consumidor acessará exatamente 1 tópico, se criarmos mais consumidores depois disso, eles ficarão ociosos, pois não existirá mais partições para serem alocadas.

Acontece a mesma coisa no caso de pararmos um consumidor, se temos 3 consumidores, cada um acessando uma partição e um dos consumidores para de funcionar, por um erro ou porque paramos de executar a aplicação, o Kafka automaticamente detectará que esse consumidor parou de funcionar e rebalanceará as partições entre os consumidores restantes.

O rebalanceamento é feito em três situações, quando um novo consumidor entra em um grupo, quando um consumidor é finalizado e avisa o Kafka que ele saiu do grupo e quando um consumidor para de responder por um longo tempo. Neste último caso, o Kafka controla quais os consumidores estão ativos utilizando um mecanismo chamado `heartbeat`. Com ele, os consumidores mandam uma mensagem para o Kafka de tempos em tempos dizendo que continuam rodando e, se essa mensagem para de ser enviada, ou se o consumidor está com algum problema e demora muito tempo para enviar essa mensagem, o Kafka assume que o consumidor parou de funcionar, e assim faz o rebalanceamento das partições.

8.2 Recriando o tópico

Quando executamos as nossas aplicações anteriormente e definimos que elas acessariam os tópicos `SHOP_TOPIC` e `SHOP_TOPIC_EVENT`. Esses tópicos foram criados automaticamente no Kafka, pois, com o Spring Boot, quando um tópico é utilizado, mas ele ainda não existe, ele é criado durante a execução da aplicação. Porém, o tópico é criado utilizando as configurações padrão de um tópico do Kafka.

Entre essas configurações está o número de partições desse tópico e, por padrão, os tópicos são criados com apenas uma partição. Como falado anteriormente, se criarmos mais de um consumidor em um mesmo grupo em um tópico com apenas uma partição, esse consumidor não conseguirá se conectar ao tópico e ele não funcionará corretamente.

Por isso, precisaremos recriar o tópico para que ele tenha mais de uma partição. Para fazer isso, vamos utilizar as ferramentas do Kafka que vimos no capítulo 2. Vamos primeiro listar os tópicos existentes no Kafka com o comando:

```
./bin/kafka-topics.sh \
    --list \
```

```
    --bootstrap-server localhost:9092
```

Esse comando deve retornar três tópicos na resposta, os dois que utilizamos na aplicação até agora, `SHOP_TOPIC` e `SHOP_TOPIC_EVENT`, e o `__consumer_offsets` que é um tópico para uso interno do Kafka. A seguir é mostrada a listagem dos tópicos:

```
SHOP_TOPIC
SHOP_TOPIC_EVENT
__consumer_offsets
```

Podemos também descrever os tópicos utilizando o seguinte comando:

```
./bin/kafka-topics.sh \
    --describe \
    --topic SHOP_TOPIC \
    --bootstrap-server localhost:9092
```

Esse comando retorna diversas informações sobre o tópico e suas partições. No exemplo a seguir, vemos na propriedade `PartitionCount` que nosso tópico possui apenas uma partição. Após as informações do tópico, vemos também as informações sobre essa partição, como o identificador da partição, no caso 0. Nos próximos exemplos, veremos que quando tivermos mais de uma partição no tópico, esse comando retornará várias linhas, descrevendo todas as partições do tópico.

```
Topic: SHOP_TOPIC    TopicId: UnfT3bEKQjOBlr00kzciew    PartitionCount: 1
ReplicationFactor: 1    Configs: segment.bytes=1073741824
    Topic: SHOP_TOPIC    Partition: 0    Leader: 0    Replicas: 0    Isr:
0
```

Vamos agora deletar esse tópico para depois recriá-lo com mais de uma partição. Para excluir um tópico, primeiro devemos alterar um arquivo de configuração do Kafka, pois por padrão (e por segurança) o Kafka desabilita a exclusão de tópicos. A mudança deve ser feita no arquivo `config/kraft/server.properties`, que está dentro da pasta do Kafka. Basta adicionar a linha `delete.topic.enable=true` a esse

arquivo (essa linha pode ser adicionada a qualquer parte do arquivo). Depois disso é necessário reiniciar o Kafka para que a mudança comece a funcionar.

Finalmente, para excluir o tópico, podemos utilizar o comando:

```
./bin/kafka-topics.sh \
    --bootstrap-server localhost:9092 \
    --delete \
    --topic SHOP_TOPIC
```

Depois, criamos novamente o tópico. Note agora a opção `--partitions` , que define quantas partições existirão para o tópico. Nesse caso eu criei o tópico com seis partições, mas poderíamos ter criado com qualquer outro número de partições.

```
./bin/kafka-topics.sh \
    --create \
    --bootstrap-server localhost:9092 \
    --partitions 6 \
    --topic SHOP_TOPIC
```

Vamos descrever o tópico novamente, utilizando o seguinte comando:

```
./bin/kafka-topics.sh \
    --describe \
    --topic SHOP_TOPIC \
    --bootstrap-server localhost:9092
```

Agora, é possível verificar que a propriedade `PartitionCount` do tópico é 6, e que existem seis partições no tópico criado.

```
Topic: SHOP_TOPIC    TopicId: Vdcv1aWIQfScbgTVW7xDEQ    PartitionCount: 6
ReplicationFactor: 1    Configs: segment.bytes=1073741824
    Topic: SHOP_TOPIC    Partition: 0    Leader: 0    Replicas: 0    Isr:
0
    Topic: SHOP_TOPIC    Partition: 1    Leader: 0    Replicas: 0    Isr:
0
    Topic: SHOP_TOPIC    Partition: 2    Leader: 0    Replicas: 0    Isr:
```

```
0
    Topic: SHOP_TOPIC    Partition: 3    Leader: 0    Replicas: 0    Isr:
0
    Topic: SHOP_TOPIC    Partition: 4    Leader: 0    Replicas: 0    Isr:
0
    Topic: SHOP_TOPIC    Partition: 5    Leader: 0    Replicas: 0    Isr:
0
```

Assim podemos subir até seis consumidores, que o Kafka distribuirá as mensagens entre esses seis consumidores. Se criarmos menos consumidores, o Kafka associará mais de uma partição aos consumidores: se tivermos apenas um consumidor, as seis partições estarão associadas ao único consumidor; com dois consumidores, o Kafka associará três partições para cada consumidor, e assim por diante.

> **QUANTAS PARTIÇÕES CRIAR?**
>
> Não existe um número certo de partições para criar, isso depende bastante da aplicação. Uma recomendação de vários utilizadores do Kafka é criar um tópico com o dobro de partições do que o número esperado de consumidores, assim temos espaço para aumentar o número de consumidores do Kafka caso seja necessário.

8.3 Executando a aplicação com mais de um consumidor

Agora podemos inicializar várias instâncias da aplicação `shop-validator`, o que é bastante simples. A única coisa que temos que mudar é a porta da aplicação cada vez que a inicializarmos e podemos mudar isso facilmente no arquivo `application.properties`.

No capítulo 5, subimos essa aplicação utilizando a porta 8081. Podemos subir uma instância nessa mesma porta e outras duas instâncias em qualquer outra porta. Por exemplo, eu utilizei as portas 8085 e 8086.

O arquivo `application.properties` da primeira instância deve ser:

```
server.port=8081
spring.jpa.hibernate.ddl-auto=none
```

O arquivo `application.properties` da segunda instância deve ser:

```
server.port=8085
spring.jpa.hibernate.ddl-auto=none
```

O arquivo `application.properties` da terceira instância deve ser:

```
server.port=8086
spring.jpa.hibernate.ddl-auto=none
```

Ao subirmos a primeira instância do `shop-validator`, vemos esta mensagem no log:

```
o.a.k.c.c.internals.ConsumerCoordinator  :
    [Consumer clientId=consumer-group-1, groupId=group]
    Adding newly assigned partitions:
    SHOP_TOPIC-0, SHOP_TOPIC-1, SHOP_TOPIC-2,
    SHOP_TOPIC-3, SHOP_TOPIC-4, SHOP_TOPIC-5
```

Essa mensagem indica que o consumidor se conectou ao tópico e, como ele é o único consumidor que existe inicialmente, todas as partições são alocadas a ele (`SHOP_TOPIC-0` , `SHOP_TOPIC-1` , `SHOP_TOPIC-2` , `SHOP_TOPIC-3` , `SHOP_TOPIC-4` , `SHOP_TOPIC-5`). Note também que o Kafka nomeia as partições usando um índice após o nome do tópico.

Quando iniciamos a segunda instância do `shop-validator`, vemos esta mensagem no log na nova instância:

```
o.s.k.l.KafkaMessageListenerContainer    :
    group: partitions assigned:
    [SHOP_TOPIC-3, SHOP_TOPIC-4, SHOP_TOPIC-5]
```

Essa mensagem indica que o consumidor foi alocado a 3 partições (`SHOP_TOPIC-3` , `SHOP_TOPIC-4` , `SHOP_TOPIC-5`).

Também vemos esta mensagem na primeira instância:

```
o.s.k.l.KafkaMessageListenerContainer    :
    group: partitions assigned:
    [SHOP_TOPIC-0, SHOP_TOPIC-1, SHOP_TOPIC-2]
```

Isso indica que a primeira instância, que antes estava associada a todas as partições, agora está alocada para 3 partições (`SHOP_TOPIC-0` , `SHOP_TOPIC-1` , `SHOP_TOPIC-2`).

Com isso, podemos notar que o Kafka rebalenceou as partições associadas a cada um dos consumidores de forma automática e transparente, sem que fosse necessária nenhuma mudança no código. Quando subirmos a terceira instância do `shop-validator` , acontecerá a mesma coisa novamente, e o Kafka vai associar duas partições para cada consumidor. Como temos seis partições, podemos subir até no máximo seis consumidores para esse tópico, assim cada partição ficará associada a uma partição.

O rebalanceamento automático das partições também acontece caso um dos consumidores pare de funcionar, ou caso o desenvolvedor queira diminuir o número de consumidores. Nesse caso, o Kafka notará que um dos consumidores não está mais disponível e rebalanceará as partições entre os consumidores que continuam funcionando, tudo isso de forma automática e transparente para o programador.

Agora que subimos as três instâncias da nossa aplicação, podemos realizar algumas compras enviando a chamada para a rota `POST /shop` . Para exemplificar, eu fiz seis compras na aplicação, que podem ser visualizadas no JSON a seguir:

```
[
    {
        "identifier": "8f144b2f-a882-4b0d-9b57-186220c356e6",
        "dateShop": "2021-10-09",
        "status": "SUCCESS",
        "items": [
            {
                "productIdentifier": "123456789",
                "amount": 1,
                "price": 1000.0
            }
        ]
    },
    {
        "identifier": "52d354a7-adea-47b9-8e29-d7009ecf0a15",
        "dateShop": "2021-10-09",
        "status": "SUCCESS",
        "items": [
            {
                "productIdentifier": "123456789",
                "amount": 1,
                "price": 1000.0
            }
        ]
    },
    {
        "identifier": "e900aced-87de-4f4a-b17f-a8dcd8261cc4",
        "dateShop": "2021-10-09",
        "status": "SUCCESS",
        "items": [
            {
                "productIdentifier": "123456789",
                "amount": 1,
                "price": 1000.0
            }
        ]
    },
    {
        "identifier": "a9306063-11be-40dc-a19e-1e094ee719d4",
```

```
        "dateShop": "2021-10-09",
        "status": "SUCCESS",
        "items": [
            {
                "productIdentifier": "123456789",
                "amount": 1,
                "price": 1000.0
            }
        ]
    },
    {
        "identifier": "fc1451e1-841e-4278-9715-37eda6c6bd3d",
        "dateShop": "2021-10-09",
        "status": "SUCCESS",
        "items": [
            {
                "productIdentifier": "123456789",
                "amount": 1,
                "price": 1000.0
            }
        ]
    },
    {
        "identifier": "35ce7da9-8175-4f38-b519-5e9018224156",
        "dateShop": "2021-10-09",
        "status": "SUCCESS",
        "items": [
            {
                "productIdentifier": "123456789",
                "amount": 1,
                "price": 1000.0
            }
        ]
    }
]
```


Podemos verificar agora no log das três instâncias do `shop-validator` quais compras foram processadas por quais consumidores. Podemos ver no log da primeira instância que ela processou as compras

e900aced-87de-4f4a-b17f-a8dcd8261cc4 e a9306063-11be-40dc-a19e-1e094ee719d4 .

```
com.santana.events.ReceiveKafkaMessage   : Compra recebida no tópico: e900aced-87de-4f4a-b17f-a8dcd8261cc4.
com.santana.events.ReceiveKafkaMessage   : Compra e900aced-87de-4f4a-b17f-a8dcd8261cc4 efetuada com sucesso.
com.santana.events.ReceiveKafkaMessage   : Compra recebida no tópico: a9306063-11be-40dc-a19e-1e094ee719d4.
com.santana.events.ReceiveKafkaMessage   : Compra a9306063-11be-40dc-a19e-1e094ee719d4 efetuada com sucesso.
```

A segunda instância processou as compras 52d354a7-adea-47b9-8e29-d7009ecf0a15 e fc1451e1-841e-4278-9715-37eda6c6bd3d .

```
com.santana.events.ReceiveKafkaMessage   : Compra recebida no tópico: 52d354a7-adea-47b9-8e29-d7009ecf0a15.
com.santana.events.ReceiveKafkaMessage   : Compra 52d354a7-adea-47b9-8e29-d7009ecf0a15 efetuada com sucesso.
com.santana.events.ReceiveKafkaMessage   : Compra recebida no tópico: fc1451e1-841e-4278-9715-37eda6c6bd3d.
com.santana.events.ReceiveKafkaMessage   : Compra fc1451e1-841e-4278-9715-37eda6c6bd3d efetuada com sucesso.
```

A terceira instância processou as compras 8f144b2f-a882-4b0d-9b57-186220c356e6 e 35ce7da9-8175-4f38-b519-5e9018224156 .

```
com.santana.events.ReceiveKafkaMessage   : Compra recebida no tópico: 8f144b2f-a882-4b0d-9b57-186220c356e6.
com.santana.events.ReceiveKafkaMessage   : Compra 8f144b2f-a882-4b0d-9b57-186220c356e6 efetuada com sucesso.
com.santana.events.ReceiveKafkaMessage   : Compra recebida no tópico: 35ce7da9-8175-4f38-b519-5e9018224156.
com.santana.events.ReceiveKafkaMessage   : Compra 35ce7da9-8175-4f38-b519-5e9018224156 efetuada com sucesso.
```

Note que, na execução que eu fiz, a distribuição acabou ficando perfeita, o que acontecerá na maioria dos casos, mas não é garantido. Por algum motivo o Kafka pode adicionar mais mensagens em uma partição do que em outras, e também é possível que, por

alguma instabilidade da aplicação, as partições sejam realocadas durante a execução da aplicação, fazendo com que alguns consumidores processem mais mensagens do que outros. Porém, a tendência é que a distribuição entre as partições e consumidores seja bem balanceada para garantir que nenhuma mensagem demore muito mais que as outras para ser processada.

Pronto! Conseguimos distribuir e paralelizar as mensagens do Kafka. Com isso, podemos acelerar o processamento das mensagens e implementar diferentes aplicações que se conectam aos tópicos do Kafka.

Porém, com as múltiplas partições, nós criamos um problema: imagine que queremos garantir que pelo menos as compras de um mesmo cliente sejam processadas na exata ordem em que as compras foram feitas. Do jeito como nossa aplicação está implementada, não há como garantir isso, pois, se um usuário enviar duas compras praticamente ao mesmo tempo, uma logo após a outra, é impossível saber em qual partição as mensagens serão alocadas e assim pode acontecer de a compra que foi enviada depois ser processada antes.

Para resolver isso, podemos utilizar o conceito de chaves do Kafka, já que mensagens que possuem a mesma chave, serão sempre alocadas em uma mesma partição quando enviadas para um mesmo tópico. Isso garante que a ordem de chegada das mensagens seja a ordem de processamento. Para isso, teremos que fazer algumas alterações em nossa aplicação. Veremos como utilizar as chaves em nossa aplicação no próximo capítulo.

Capítulo 9
Usando chaves nas mensagens

Em grande parte das aplicações, a ordem das mensagens não é importante. Nesses casos não precisamos definir as chaves no Kafka, pois assim as mensagens serão alocadas em qualquer partição e o processamento será em qualquer ordem. Porém, em algumas aplicações, a ordem das mensagens pode ser um requisito essencial, como nos casos de ordens de operações bancárias ou no fluxo de um workflow.

Nesses casos, temos duas opções para garantir a ordem das mensagens do Kafka: a primeira opção é utilizar apenas uma partição, o que não é o ideal, pois com apenas uma partição não conseguiremos processar as mensagens em paralelo e perderemos todas as vantagens que vimos no capítulo anterior.

A segunda forma é definir chaves para as mensagens, com elas, o Kafka garante que todas as mensagens que compartilham a mesma chave sejam enviadas para o mesmo tópico para serem alocadas sempre na mesma partição.

9.1 Adicionando chaves nas mensagens

Precisaremos mudar a aplicação para adicionar uma chave a todas as mensagens do Kafka que enviaremos para o tópico `SHOP_TOPIC` no produtor. Essa chave garantirá que as mensagens feitas pelo mesmo comprador sejam enviadas sempre para o mesmo tópico. O primeiro passo nessa mudança é adicionar a coluna `buyer_identifier` no script do banco de dados da `shop-api` . Já criamos esse script no arquivo

`schema.sql` no capítulo 3, agora temos apenas que adicionar essa nova coluna. O código SQL a seguir mostra como deve ficar o novo script:

```
create table shop (
    id bigint generated by default as identity,
    buyer_identifier varchar(100) not null,
    identifier varchar not null,
    status varchar not null,
    date_shop date,
    primary key (id)
);

create table shop_item (
    id bigint generated by default as identity,
    product_identifier varchar(100) not null,
    amount int not null,
    price float not null,
    shop_id bigint REFERENCES shop(id),
    primary key (id)
);
```

Além disso, temos que adicionar o campo `buyerIdentifier` às classes `ShopDTO` e `Shop`. Enviaremos esse novo campo na chamada da rota `POST /shop` e ele será salvo no banco de dados, na tabela `shop`. O código a seguir mostra as mudanças na classe `ShopDTO`.

```
@Getter
@Setter
public class ShopDTO {
    private String identifier;
    private LocalDate dateShop;
    private String status;
    private String buyerIdentifier; // novo campo
    private List<ShopItemDTO> items = new ArrayList<>();

    public static ShopDTO convert(Shop shop) {
        ShopDTO shopDTO = new ShopDTO();
        shopDTO.setIdentifier(shop.getIdentifier());
```

```
        shopDTO.setDateShop(shop.getDateShop());
        shopDTO.setStatus(shop.getStatus());
        shopDTO.setBuyerIdentifier(shop.getBuyerIdentifier());
        shopDTO.setItems(shop
                    .getItems()
                    .stream()
                    .map(i -> ShopItemDTO.convert(i))
                    .collect(Collectors.toList()));
        return shopDTO;
    }

}
```

O código a seguir mostra as mudanças na classe `Shop`.

```
@Getter
@Setter
@Entity(name="shop")
public class Shop {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String identifier;

    private String status;

    @Column(name = "date_shop")
    private LocalDate dateShop;

    @Column(name = "buyer_identifier")
    private String buyerIdentifier;

    @OneToMany(fetch = FetchType.EAGER,
        cascade = CascadeType.ALL,
        mappedBy = "shop")
    private List<ShopItem> items;

    public static Shop convert(ShopDTO shopDTO) {
```

```
        Shop shop = new Shop();
        shop.setIdentifier(shopDTO.getIdentifier());
        shop.setStatus(shopDTO.getStatus());
        shop.setDateShop(shopDTO.getDateShop());
        shop.setItems(shopDTO
                .getItems()
                .stream()
                .map(i -> ShopItem.convert(i))
                .collect(Collectors.toList()));
        shop.setBuyerIdentifier(shopDTO.getBuyerIdentifier());
        return shop;
    }

}
```

Agora que o campo já está disponível no DTO e na classe do modelo, podemos utilizá-lo na chamada para o Kafka. A mudança será bastante simples, basta adicionar a chave na chamada do método `send` do `kafkaTemplate`. Essa mudança deve ser feita na classe `KafkaClient`, que implementamos no capítulo 4. O código a seguir mostra o código completo da classe:

```
@Service
@RequiredArgsConstructor
public class KafkaClient {

    private final KafkaTemplate<String, ShopDTO> kafkaTemplate;

    private static final String SHOP_TOPIC_NAME = "SHOP_TOPIC";

    public void sendMessage(ShopDTO msg) {
        kafkaTemplate.send(
            SHOP_TOPIC_NAME,
            msg.getBuyerIdentifier(),
            msg);
    }

}
```

A única mudança que fizemos nesse método foi adicionar o parâmetro `msg.getBuyerIdentifier()`, que será a chave da mensagem, à chamada do método `send`. Com isso, o Kafka receberá a chave da mensagem, que é o identificador do comprador, e alocará todas as mensagens que tenham o mesmo comprador nas partições corretas.

9.2 Recebendo as chaves nas mensagens

A chave é importante principalmente para que o Kafka consiga alocar as mensagens nas partições corretas, mas podemos alterar a aplicação `shop-validator` para receber a chave enviada no consumidor. Isso não é obrigatório, pois, se executarmos a `shop-validator` do jeito que ela estava antes, tudo continuaria funcionando exatamente da mesma maneira, apenas o funcionamento interno do Kafka seria alterado.

Como exemplo, vamos alterar a aplicação para receber a chave no consumidor e verificar que também podemos receber outras informações do Kafka, como a partição em que a mensagem está e o horário em que a mensagem foi recebida no tópico.

A primeira mudança que temos que fazer é adicionar o campo `buyerIdentifier` na classe `ShopDTO` - exatamente a mesma mudança que fizemos na `shop-api`. O código a seguir mostra como essa classe deve ficar com o campo novo:

```
@Getter
@Setter
public class ShopDTO {
    private String identifier;
    private LocalDate dateShop;
    private String status;
    private String buyerIdentifier;
    private List<ShopItemDTO> items;
}
```

Depois, apenas temos que alterar o método `listenShopTopic` da classe `ReceiveKafkaMessage` para adicionar alguns parâmetros novos ao método `listenShopTopic` . Esses parâmetros receberão a chave da mensagem, o identificador da partição que recebeu a mensagem e o horário em que a mensagem foi recebida. Com isso, poderemos verificar se as chaves funcionaram, pois mensagens com a mesma chave obrigatoriamente devem ser processadas pelo mesmo consumidor.

```
@KafkaListener(topics = SHOP_TOPIC_NAME, groupId = "group")
public void listenShopTopic(ShopDTO shopDTO,
        @Header(KafkaHeaders.RECEIVED_MESSAGE_KEY)
            String key,
        @Header(KafkaHeaders.RECEIVED_PARTITION_ID)
            String partitionId,
        @Header(KafkaHeaders.RECEIVED_TIMESTAMP)
            String timestamp) {

        log.info("Compra recebida no tópico: {} " +
            "com chave {} na partição {} hora {}.",
               shopDTO.getIdentifier(),
               key, partitionId, timestamp);

    // continuação do método

}
```

9.3 Executando a aplicação

Podemos subir a aplicação novamente como fizemos no capítulo anterior, com três instâncias do `shop-validator` , assim poderemos verificar se todas as compras de um mesmo comprador foram enviadas para a mesma partição do Kafka. O JSON enviado para a rota `POST /shop` agora deve conter também o campo `buyerIdentifier` ,

que será utilizado como chave para a mensagem do Kafka. O JSON a seguir mostra um exemplo dos dados que devem ser enviados para a rota:

```
{
    "buyerIdentifier": "b-1",
    "items": [
        {
            "productIdentifier": "123456789",
            "amount": "1",
            "price": "1000"
        }
    ]
}
```

No meu teste, eu mandei nove mensagens para o Kafka, sendo três para cada `buyerIdentifier` diferente, b-1, b-2 e b-3. Vejam nos logs a seguir que todas as mensagens que tinham a mesma chave foram alocadas sempre na mesma partição e, por isso, foram processadas sempre pelo mesmo consumidor. As primeiras mensagens, que tiveram a chave `b-1` foram todas alocadas na partição 2.

```
com.santana.events.ReceiveKafkaMessage   : Compra recebida no tópico:
21c9a4a7-50c9-427b-b280-767281d7b229 com chave b-1 na partição 2 hora
1634066955214.
com.santana.events.ReceiveKafkaMessage   : Compra 21c9a4a7-50c9-427b-b280-
767281d7b229 efetuada com sucesso.
com.santana.events.ReceiveKafkaMessage   : Compra recebida no tópico:
b2ca2523-3e05-4286-aa00-a6cdb33233d9 com chave b-1 na partição 2 hora
1634066955858.
com.santana.events.ReceiveKafkaMessage   : Compra b2ca2523-3e05-4286-aa00-
a6cdb33233d9 efetuada com sucesso.
com.santana.events.ReceiveKafkaMessage   : Compra recebida no tópico:
16afa9f5-0b81-4fca-80cd-19751ada287d com chave b-1 na partição 2 hora
1634066956519.
com.santana.events.ReceiveKafkaMessage   : Compra 16afa9f5-0b81-4fca-80cd-
19751ada287d efetuada com sucesso.
```

As mensagens com chave `b-2` foram todas alocadas na partição 0.

```
com.santana.events.ReceiveKafkaMessage   : Compra recebida no tópico:
4b50f969-10f5-4508-a3c6-070533f8e0f8 com chave b-2 na partição 0 hora
1634067032736.
com.santana.events.ReceiveKafkaMessage   : Compra 4b50f969-10f5-4508-a3c6-
070533f8e0f8 efetuada com sucesso.
com.santana.events.ReceiveKafkaMessage   : Compra recebida no tópico:
8c6491c6-3138-4d5c-bdc4-4ae6d5d8d6ca com chave b-2 na partição 0 hora
1634067033264.
com.santana.events.ReceiveKafkaMessage   : Compra 8c6491c6-3138-4d5c-bdc4-
4ae6d5d8d6ca efetuada com sucesso.
com.santana.events.ReceiveKafkaMessage   : Compra recebida no tópico:
81790257-71a6-4075-9954-332e171e6734 com chave b-2 na partição 0 hora
1634067033773.
com.santana.events.ReceiveKafkaMessage   : Compra 81790257-71a6-4075-9954-
332e171e6734 efetuada com sucesso.
```

Finalmente, as mensagens com chave `b-3` foram também alocadas na partição 0.

```
com.santana.events.ReceiveKafkaMessage   : Compra recebida no tópico:
8c3df85b-7b97-4d73-9b3e-67db370cceaf com chave b-3 na partição 0 hora
1634067090119.
com.santana.events.ReceiveKafkaMessage   : Compra 8c3df85b-7b97-4d73-9b3e-
67db370cceaf efetuada com sucesso.
com.santana.events.ReceiveKafkaMessage   : Compra recebida no tópico:
4cc4fdb7-8a62-4cb3-9a7b-7199f5c7e0f7 com chave b-3 na partição 0 hora
1634067090630.
com.santana.events.ReceiveKafkaMessage   : Compra 4cc4fdb7-8a62-4cb3-9a7b-
7199f5c7e0f7 efetuada com sucesso.
com.santana.events.ReceiveKafkaMessage   : Compra recebida no tópico:
ebbaf0eb-afc4-494c-9af2-bb08d3f379ad com chave b-3 na partição 0 hora
1634067091156.
com.santana.events.ReceiveKafkaMessage   : Compra ebbaf0eb-afc4-494c-9af2-
bb08d3f379ad efetuada com sucesso.
```

Note que as chaves causam um grande desbalanceamento no uso das partições, pois a partição 0 recebeu 6 mensagens, a partição 2 recebeu 3 mensagens e a partição 1 não recebeu nenhuma mensagem. Isso aconteceu por causa das chaves, pois todas as

mensagens com a mesma chave foram alocadas na partição onde a primeira mensagem com aquela chave foi alocada, independentemente de isso causar um desbalanceamento nas partições ou não. Por isso, as chaves nas mensagens do Kafka só devem ser utilizadas quando a ordem da mensagem for extremamente importante; caso contrário, evite usar as chaves.

Uma nota importante: a aplicação `shop-report` também consome mensagens do tópico `SHOP_TOPIC`, então ela também pode receber as chaves do Kafka. Mas, como ela não usará essa informação, não precisamos mudar nada na implementação dessa aplicação, já que ela continuará recebendo as mensagens exatamente como antes. Ela apenas ignorará a chave que vem na mensagem do Kafka.

Vimos que o conceito de chaves é bastante útil quando precisamos garantir a ordem de processamento das mensagens. Vimos também que adicionar as chaves à mensagem é bastante simples, sem alterar muito a implementação das aplicações.

Vimos como distribuir e como adicionar as chaves nas mensagens, mas ainda não vimos como tratar erros nos processamentos das mensagens. Os erros podem acontecer por diversos motivos, sejam erros nos dados enviados, na implementação da aplicação, ou erros no ambiente de execução da aplicação. Por isso, é sempre importante que as aplicações estejam preparadas para se recuperar de possíveis falhas.

No Kafka, fazer retentativas de processamento das mensagens é essencial para garantir que mensagens não sejam perdidas por erros que podem ser corrigidos. No próximo capítulo, veremos como fazer o reprocessamento de mensagens que tiveram problemas no seu processamento.

Capítulo 10
Retentativas

Uma funcionalidade importante do Kafka é fazer retentativas no consumo das mensagens, isso porque o processamento de uma mensagem pode falhar por algum problema na aplicação. Do jeito como a nossa aplicação está implementada, estamos usando o mecanismo padrão de reprocessamento para o Kafka, mas veremos que ele pode fazer o processamento das mensagens ficar bastante lento, pois ele tentará reprocessar uma mensagem várias vezes antes de continuar lendo as mensagens do tópico.

Para ver como as retentativas podem ser implementadas, vamos criar uma aplicação nova, chamada `shop-retry`, que tentará filtrar as compras que possuem algum problema no processamento, por exemplo, compras sem produtos.

Após verificar como o mecanismo padrão de retentivas funciona, para não ficar dependente apenas dele, vamos também implementar um outro mecanismo, que criará um novo tópico que será responsável apenas por tentar processar mensagens que tiveram erros, liberando o tópico principal para processar apenas as mensagens que não tiveram erros, para que seu consumidor não fique bloqueado.

10.1 Configuração da aplicação

Para a aplicação, precisaremos adicionar apenas a dependência do Spring Boot, do Kafka e do Lombok para implementar um consumidor, como já fizemos no capítulo 5. O código a seguir mostra o arquivo `pom.xml` criado para essa aplicação.

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    https://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <groupId>com.santana</groupId>
    <artifactId>shop-retry</artifactId>
    <version>1.0-SNAPSHOT</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.3.5</version>
    </parent>

    <properties>
        <maven.compiler.source>21</maven.compiler.source>
        <maven.compiler.target>21</maven.compiler.target>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.kafka</groupId>
            <artifactId>spring-kafka</artifactId>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.34</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>

</project>
```

Como todas as aplicações Spring Boot, sempre precisamos criar a classe com o método `Main`, como mostrado no código a seguir.

```
package com.santana;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure
    .SpringBootApplication;

@SpringBootApplication
public class Main {

    public static void main(String [] args) {
        SpringApplication.run(Main.class, args);
    }

}
```

Também precisamos criar a classe de configuração do Kafka, a `KafkaConfig`, como foi feito no capítulo 4. Não vou colocar o código aqui, pois ele é exatamente igual ao que está lá, basta copiar a classe que está lá e colocar na aplicação nova.

10.2 Implementando o consumidor

Vamos implementar um primeiro consumidor que apenas lê as mensagens que são recebidas do tópico `SHOP_TOPIC_EVENT` e depois sempre lança uma exceção para a compra que não possui itens cadastrados. Faremos isso para mostrar que, mesmo no caso em que o consumidor lança uma exceção, o Kafka tentará sempre reprocessar a mensagem, pois quando o consumidor retorna uma exceção, o Kafka entende que a mensagem não foi processada com sucesso.

```
package com.santana.events;

import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Service;
```

```
import com.santana.dto.ShopDTO;

import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;

@Slf4j
@Service
@RequiredArgsConstructor
public class ReceiveKafkaMessage {

    private static final String SHOP_TOPIC = "SHOP_TOPIC";

    @KafkaListener(
        topics = SHOP_TOPIC,
        groupId = "group_report")
    public void listenShopTopic(ShopDTO shopDTO)
        throws Exception {
        log.info("Compra recebida no tópico: {}.",
            shopDTO.getIdentifier());
        if (shopDTO.getItems() == null ||
                shopDTO.getItems().isEmpty()) {
            log.error("Compra sem items.");
            throw new Exception();
        }
    }
}
```

Vamos agora executar essa aplicação. Note que ao recebermos a primeira compra foi lançada uma exceção, de que o consumidor tentou novamente processar a mesma mensagem diversas vezes. Isso acontecerá 10 vezes seguidas, pois, por padrão, um consumidor tenta reprocessar uma mensagem que deu erro 10 vezes. Note no log a seguir os valores `currentAttempts`, que mostra em qual tentativa está o processamento da imagem, no caso, na nona tentativa, e o `maxAttempts`, que é o número máximo de tentativas que o Kafka fará para processar a mensagem.

```
021-12-05 19:28:50.527  INFO 11129 --- [ntainer#0-0-C-1]
com.santana.events.ReceiveKafkaMessage
```

```
    : Compra recebida no tópico: 2c0f8495-61b8-4f65-982c-bd0c87dae657.
2021-12-05 19:28:50.528 ERROR 11129 --- [ntainer#0-0-C-1]
o.s.k.l.SeekToCurrentErrorHandler :
    Backoff FixedBackOff{interval=0, currentAttempts=9, maxAttempts=9}
    exhausted for ConsumerRecord(topic = SHOP_TOPIC, partition = 0,
    leaderEpoch = 0, offset = 12, CreateTime = 1638743325995, serialized
    key size = 3, serialized value size = 189, headers =
RecordHeaders(headers = [],
    isReadOnly = false), key = b-3, value =
com.santana.dto.ShopDTO@4a6bc09a)
```

Esse tipo de reprocessamento funciona, mas existe um problema nessa implementação: o consumidor fica travado na mensagem que deu erro por 10 tentativas seguidas, pois as próximas mensagens não serão lidas enquanto a mensagem com erro não for descartada pelo consumidor. Precisamos implementar então um mecanismo que libere o consumidor para ler as próximas mensagens, enquanto a mensagem é reprocessada em outro lugar.

Vamos melhorar um pouco a nossa implementação, para que seja feito o tratamento de exceções utilizando o `try catch`. O código a seguir mostra algumas alterações feitas no método `listenShopTopic`.

```
@KafkaListener(topics = SHOP_TOPIC, groupId = "group_report")
public void listenShopTopic(ShopDTO shopDTO) throws Exception {
    try {
        log.info("Compra recebida no tópico: {}.",
            shopDTO.getIdentifier());
        if (shopDTO.getItems() == null ||
                shopDTO.getItems().isEmpty()) {
            log.error("Compra sem items");
            throw new Exception();
        }
    } catch(Exception e) {
        log.info("Erro na aplicação");
    }
}
```

Podemos rodar a nossa aplicação assim, mas agora veremos que as mensagens não serão reprocessadas. Isso porque, para o Kafka, como não está sendo retornada uma exceção, agora as mensagens foram lidas corretamente.

```
com.santana.events.ReceiveKafkaMessage   :
    Compra recebida no tópico: b104f3f8-1c1b-4f38-85e4-9da6861bf8ab.
com.santana.events.ReceiveKafkaMessage   :
    Compra sem items
com.santana.events.ReceiveKafkaMessage   :
    Erro na aplicação
com.santana.events.ReceiveKafkaMessage   :
    Compra recebida no tópico: d930f5e8-f334-46b2-9e96-048507991897.
com.santana.events.ReceiveKafkaMessage   :
    Compra sem items
com.santana.events.ReceiveKafkaMessage   :
    Erro na aplicação
com.santana.events.ReceiveKafkaMessage   :
    Compra recebida no tópico: 8f6cadd1-ba33-4ed8-a1b0-bf195f47eb29.
com.santana.events.ReceiveKafkaMessage   :
    Compra sem items
com.santana.events.ReceiveKafkaMessage   :
    Erro na aplicação
```

10.3 Melhorando as retentativas

O que vamos fazer agora é, em vez de apenas escrever uma mensagem no log indicando que deu um erro no processamento da mensagem, vamos também enviar a mensagem original para um novo tópico, que chamaremos de `SHOP_TOPIC_RETRY`. Todas as mensagens que tiverem algum erro no processamento serão enviadas para esse tópico. Depois, vamos implementar um novo consumidor que será responsável apenas por tentar reprocessar as mensagens que tiveram falha na primeira tentativa de processamento.

Vamos mudar o método `listenShopTopic` para fazer o processamento que comentamos, então no `catch` adicionamos uma linha para que a mensagem seja enviada para outro tópico quando haver um problema no processamento desta mensagem. Assim, o consumidor não ficará parado tentando reprocessar as mensagens.

```
private final KafkaTemplate<String, ShopDTO> kafkaTemplate;

private static final String SHOP_TOPIC
    = "SHOP_TOPIC";
private static final String SHOP_TOPIC_RETRY
    = "SHOP_TOPIC_RETRY";

@KafkaListener(topics = SHOP_TOPIC,
    groupId = "group_retry")
public void listenShopTopic(ShopDTO shopDTO)
    throws Exception {
    try {
        if (shopDTO.getItems() == null ||
                shopDTO.getItems().isEmpty()) {
            log.error("Compra sem items");
            throw new Exception();
        }
    } catch(Exception e) {
        log.info("Erro na aplicação");
        kafkaTemplate.send(SHOP_TOPIC_RETRY, shopDTO);
    }
}
```

Podemos implementar agora o consumidor desse novo tópico que criamos, que será responsável por tentar reprocessar as mensagens que tiveram erro. Esse método pode ser implementado na mesma classe do método anterior, pois o Spring Boot considerará a anotação `KafkaListener`. Como existirão dois métodos com essa anotação, os dois funcionarão como consumidores independentes, mesmo que tenham sido implementados na mesma classe.

```
@KafkaListener(topics = SHOP_TOPIC_RETRY,
     groupId = "group_retry")
```

```
public void listenShopTopicRetry(ShopDTO shopDTO)
        throws Exception {
    log.info("Retentativa de processamento: {}.",
        shopDTO.getIdentifier());
}
```

Se executarmos a aplicação agora, veremos que o primeiro consumidor tentará processar a mensagem, mas ocorrerá um erro no processamento dela, então a mensagem será colocada no tópico de retentativas. Esse tópico então processará essa mensagem.

```
com.santana.events.ReceiveKafkaMessage:
    Compra recebida no tópico: e0b6ad77-1e90-4c1e-a219-64818fa3ea84.
com.santana.events.ReceiveKafkaMessage:
    Erro na aplicação
com.santana.events.ReceiveKafkaMessage:
    Retentativa de processamento: e0b6ad77-1e90-4c1e-a219-64818fa3ea84.
com.santana.events.ReceiveKafkaMessage:
    Compra recebida no tópico: a7d314c4-f1ab-4645-9109-2b6e59ed8c38.
com.santana.events.ReceiveKafkaMessage:
    Erro na aplicação
com.santana.events.ReceiveKafkaMessage:
    Retentativa de processamento: a7d314c4-f1ab-4645-9109-2b6e59ed8c38.
```

Esse tipo de tratamento de retentativas pode ajudar muito no reprocessamento de mensagens que ocorreram por falhas inesperadas na aplicação. A maior vantagem dessa implementação é que ela libera o consumidor do tópico principal para continuar processando as mensagens, enquanto os erros são tratados por outro consumidor, que normalmente não tem tanto problema se ele ficar travado um tempo tentando processar uma mensagem.

Até agora utilizamos sempre o `kafka-cli` para administrar o Kafka, porém às vezes precisamos fazer isso dentro do código-fonte da aplicação. No próximo capítulo, veremos a biblioteca `kafka-clients`, que permite que façamos a administração do Kafka, realizando ações como criar e listar tópicos, criar e listar `consumer groups`, entre outros, diretamente na nossa aplicação.

Capítulo 11
Administrando o Kafka no Java

Nos últimos capítulos, utilizamos o `kafka-cli`, a ferramenta de linha de comando, quando precisamos gerenciar o Kafka. Porém, em algumas situações precisaremos administrar o Kafka em nossa aplicação, por exemplo, se precisarmos verificar se um tópico existe ou não para tomar uma ação, ou se o número de partições de um tópico for controlado por algum dado externo que não teremos na inicialização do Kafka. Tudo isso precisará ser feito em nossa aplicação e, apesar de ser possível utilizar o `kafka-cli` dentro da aplicação, seria bastante complexo, pois dificilmente conseguiríamos verificar se o comando foi executado com sucesso ou não.

Para resolver esse problema, temos a biblioteca `kafka-clients`, que possibilita a administração do Kafka programaticamente. Faremos uma aplicação simples, que será executada diretamente da linha de comando, mas, se for necessário, também é possível colocar esse código em uma API. Veremos como fazer diversas operações no Kafka como listar, descrever, criar e excluir tópicos e grupos de consumidores.

11.1 Configuração da aplicação

Para implementar essa aplicação, eu criei um novo projeto, que eu chamei de `kafka-manager`. Como a aplicação é bem simples, a única configuração necessária é criar o arquivo `pom.xml`, que adiciona a dependência `kafka-clients`. A listagem a seguir mostra esse arquivo.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    https://maven.apache.org/xsd/maven-4.0.0.xsd">
```

```
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.santana</groupId>
  <artifactId>kafka-manager</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <dependencies>
    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-clients</artifactId>
        <version>3.8.0</version>
    </dependency>
  </dependencies>
</project>
```

Essa aplicação tem uma classe com o método `main`, e esse método chama as funções que farão as operações no Kafka. Esse método tem duas ações importantes, a primeira é criar o objeto `properties`, no qual podem ser adicionadas as configurações para a conexão com o Kafka. Neste exemplo, a única configuração que precisamos é o endereço do Kafka, que no nosso caso é o `localhost:9092`. A segunda ação é a criação do objeto `adminClient` que fará as chamadas aos métodos que acessarão o Kafka. O código a seguir mostra a criação dessa classe.

```java
package com.santana.kafka.admin;

import java.util.Properties;
import java.util.concurrent.ExecutionException;

import org.apache.kafka.clients.admin.AdminClient;
import org.apache.kafka.clients.admin.AdminClientConfig;

public class Main {

    public static void main(String[] args)
            throws InterruptedException, ExecutionException {

        Properties properties = new Properties();
        properties.put(
                AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG,
                "localhost:9092");

        AdminClient adminClient
            = AdminClient.create(properties);

        KafkaAdmin.create("topico-1",
                2, (short) 1, adminClient);

        KafkaAdmin.create("topico-2",
                2, (short) 1, adminClient);

        KafkaAdmin.list(adminClient);

        KafkaAdmin.describe("topico-1", adminClient);

        KafkaAdmin.delete("topico-1", adminClient);

        KafkaAdmin.delete("topico-2", adminClient);

        KafkaAdmin.listCG(adminClient);

        KafkaAdmin.deleteCG("group", adminClient);
```

```
        KafkaAdmin.describeCluster(adminClient);

    }

}
```

Note que nessa classe estamos chamando vários métodos para gerenciar o Kafka: o que cria, lista e exclui tópicos, o que lista e exclui os `consumer groups` e um que descreve o cluster do Kafka. Vamos ver passo a passo a criação de cada um desses métodos agora.

11.2 Administrando o Kafka

Todos os métodos para gerenciar o Kafka foram implementados na classe `KafkaAdmin`. Eles basicamente chamam um outro método da classe `AdminClient`, que pertence à biblioteca `kafka-clients`. O código a seguir mostra a estrutura da classe `KafkaAdmin` com apenas os `imports` que serão necessários e a declaração da classe. Nela, implementaremos os vários métodos que administrarão o Kafka.

```
package com.santana.kafka.admin;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ExecutionException;

import org.apache.kafka.clients.admin.AdminClient;
import org.apache.kafka.clients.admin.CreateTopicsResult;
import org.apache.kafka.clients.admin.DeleteConsumerGroupsResult;
import org.apache.kafka.clients.admin.DeleteTopicsResult;
import org.apache.kafka.clients.admin.DescribeClusterResult;
import org.apache.kafka.clients.admin.DescribeTopicsResult;
import org.apache.kafka.clients.admin.ListConsumerGroupsResult;
import org.apache.kafka.clients.admin.ListTopicsResult;
import org.apache.kafka.clients.admin.NewTopic;
```

```
public class KafkaAdmin {

    // métodos que vão acessar o Kafka

}
```

Listando os tópicos

O primeiro método, chamado `list`, é o que lista os tópicos existentes no Kafka. Ele apenas chama o método que lista os tópicos do objeto `adminClient`, que chama `listTopics`. Os tópicos são retornados em um objeto da classe `ListTopicsResult`, com isso, podemos apenas iterar na lista de tópicos para imprimir o nome de todos os tópicos existentes.

```
public static void list(AdminClient adminClient)
        throws InterruptedException, ExecutionException {

    ListTopicsResult topics =
        adminClient.listTopics();

    topics
        .names()
        .get()
        .forEach(System.out::println);
}
```

O resultado da execução desse método é apenas uma saída no console com o nome de todos os tópicos que existem no Kafka, note que ele retornou os dois tópicos que criamos agora na execução do Main da aplicação `kafka-manager` e também os tópicos que foram criados nos capítulos anteriores, o `SHOP_TOPIC` e o `SHOP_TOPIC_EVENT`.

```
topico-2
topico-1
SHOP_TOPIC
SHOP_TOPIC_EVENT
```

Criando um tópico

O segundo método é o que cria um tópico no Kafka, o `create`. O método recebe como parâmetro o nome do tópico, o número de partições e o número de replicações do tópico. Depois, é feita a configuração de um novo tópico usando um objeto da classe `NewTopic`. Esse tópico é adicionado a uma lista, pois o método `createTopics` da classe `AdminClient` precisa receber sempre uma lista, mesmo que se queira criar apenas um tópico.

Quando executamos o método para a criação do tópico, ele retorna um objeto do tipo `CreateTopicsResult` que terá o resultado da execução do comando de criação do tópico, que é basicamente se o tópico foi criado com sucesso ou não.

```
public static void create(String topicName,
        int partitions,
        short replications,
        AdminClient adminClient) {

    final NewTopic newTopic =
            new NewTopic(topicName, partitions, replications);
    List<NewTopic> topics = new ArrayList<NewTopic>();

    topics.add(newTopic);

    try {
        final CreateTopicsResult result =
                adminClient.createTopics(topics);

        result
            .all()
            .get();
    } catch (final Exception e) {
        throw new RuntimeException("Failed to create topic:"
                + topicName, e);
    }

}
```

Esse método não tem nenhuma saída, ele escreverá alguma mensagem na tela apenas se a criação do tópico falhar.

Descrevendo um tópico

O terceiro método, o `describe`, descreve um tópico, trazendo informações como o identificador de um tópico e as suas partições. Esse método recebe como parâmetro o nome do tópico que queremos e esse nome deve ser adicionado a uma lista. Com essa lista, chamamos o método `describeTopics` da classe `AdminClient`. Essa classe retorna uma lista com as informações dos tópicos em um objeto do tipo `DescribeTopicsResult`, assim podemos iterar nesses resultados e imprimir as informações dos tópicos.

```
public static void describe(String topicName,
        AdminClient adminClient)
        throws InterruptedException, ExecutionException {

    List<String> topicNames = new ArrayList<>();
    topicNames.add(topicName);

    DescribeTopicsResult topics =
        adminClient.describeTopics(topicNames);

    topics
        .all()
        .get()
        .forEach(
            (x, y) -> System.out.println(
                x + " " + y.topicId() + " " + y.partitions()));
}
```

Os dados que são mostrados são o nome do tópico, o identificador do tópico, que é um valor alfanumérico gerado pelo Kafka, e as informações sobre as partições do Kafka. Por exemplo, a descrição do tópico chamado `topico-1` será:

```
topico-1 prq85m7VRZqT9ji6DwWKFw [
    (partition=0, leader=eduardo:9092 (id: 0 rack: null),
```

```
        replicas=eduardo:9092 (id: 0 rack: null),
        isr=eduardo:9092 (id: 0 rack: null)),
    (partition=1, leader=eduardo:9092 (id: 0 rack: null),
        replicas=eduardo:9092 (id: 0 rack: null),
        isr=eduardo:9092 (id: 0 rack: null))
]
```

Excluindo um tópico

O quarto método será para a exclusão de tópicos, o `delete`. Ele deve receber o nome do tópico que queremos excluir e, com isso, chamaremos o método `deleteTopics` da classe `AdminClient`. Assim como o método `create`, o `delete` também não terá nenhum retorno caso a exclusão do tópico funcione corretamente. Caso aconteça algum erro na criação do tópico, será mostrada uma mensagem e uma exceção será retornada.

```
public static void delete(String topicName, AdminClient adminClient)  {

    List<String> topicNames = new ArrayList<>();
    topicNames.add(topicName);
    try {
        DeleteTopicsResult topics =
            adminClient.deleteTopics(topicNames);

        topics
            .all()
            .get();
    } catch (final Exception e) {
        throw new RuntimeException("Failed to delete topic:"
                + topicName, e);
    }
}
```

Listando Consumer Groups

Além das operações com os tópicos, podemos gerenciar os `consumer groups`. O quinto método, o `listCG`, lista todos os grupos que existem no Kafka e chama apenas o método `listConsumerGroups` da

classe `AdminClient`. Depois disso, a lista com os grupos é retornada em um objeto da classe `ListConsumerGroupsResult` e podemos iterar sobre essa lista para mostrar os dados de todos os grupos.

```
public static void listCG(AdminClient adminClient)
        throws InterruptedException, ExecutionException {

    ListConsumerGroupsResult cgs =
        adminClient.listConsumerGroups();

    cgs
        .all()
        .get()
        .forEach(cg ->
            System.out.println(cg.groupId()));
}
```

O resultado desse método será a impressão das informações dos grupos. No nosso caso, existem dois grupos no Kafka, os que foram criados quando subimos as aplicações nos capítulos anteriores, o `group_report`, e o `group`.

```
group_report
group
```

Deletando Consumer Groups

No sexto método, o `deleteCG`, vamos deletar um `consumer group`. Nele, precisamos receber o identificador do grupo que queremos excluir, depois criamos uma lista com esse grupo e, por fim, chamamos o método `deleteConsumerGroups` da classe `AdminClient`.

```
public static void deleteCG(
    String groupId, AdminClient adminClient)
        throws InterruptedException, ExecutionException {

    List<String> groups = new ArrayList<>();
    groups.add(groupId);
```

```
    try {

        DeleteConsumerGroupsResult cgs
            = adminClient.deleteConsumerGroups(groups);
        cgs
            .all()
            .get();

    } catch (final Exception e) {
        throw new RuntimeException("Failed to delete cg:"
                + groupId, e);
    }
}
```

Esse método também não terá nenhum retorno caso o `consumer group` seja excluído com sucesso, caso contrário, será mostrada uma mensagem indicando que houve algum erro na exclusão do tópico.

Descrevendo o Cluster

É também possível recuperar informações sobre o cluster do Kafka utilizando o método `describeCluster`.

```
public static void describeCluster(AdminClient adminClient)
    throws InterruptedException, ExecutionException  {

        DescribeClusterResult cluster =
            adminClient.describeCluster();

        System.out.println(cluster.clusterId().get());
}
```

Esse método retornará o id do cluster.

```
YiiwOEzGSXurw85-eSK23w
```

Essa biblioteca possui diversos outros métodos que podem ser utilizados. Por exemplo, é possível também deletar registros nos

tópicos, criar regras de acesso aos tópicos, alterar o número de partições em tópicos. A biblioteca pode ser bem útil para casos em que precisamos saber como o Kafka está funcionando dentro da aplicação. Com certeza o caso de uso mais comum é verificar se um tópico existe ou não, se ele tem um número de partições suficientes, e criar ou excluir tópicos quando necessário. Mas há muitos outros métodos que valem a pena serem explorados quando precisamos de um controle maior do Kafka.

Todo o código apresentado até aqui foi desenvolvido em Java, mas uma outra grande característica do Kafka é a independência de linguagem. Como podemos passar mensagens no formato JSON para o tópico, podemos ler o tópico em aplicações desenvolvidas em qualquer linguagem que tenha um cliente do Kafka disponível. Como exemplo, no próximo capítulo, vamos implementar uma aplicação em Python que também acessa o tópico `SHOP_TOPIC_EVENT` que estamos utilizando em nossas aplicações escritas em Java.

Capítulo 12
Conectando no Kafka com Python

Uma das características mais interessante do Kafka é que ele é independente de linguagem. Isso quer dizer que ele pode intermediar a comunicação entre aplicações desenvolvidas em diferentes linguagens. Por exemplo, podemos enviar uma mensagem para um tópico em uma aplicação implementada em Java e depois receber essa mensagem em uma aplicação implementada em Python ou em JavaScript. Além disso, como as mensagens do Kafka podem ser enviadas no formato `JSON`, a criação de objetos pode ser transparente de uma linguagem para outra.

O Kafka possui bibliotecas implementados para diferentes linguagens, como Java, Python, Go e Ruby, e aplicações em todas essas linguagens podem trocar informações de forma bastante simples. Para exemplificar isso, neste capítulo, vamos mostrar a implementação de uma aplicação que se conectará ao tópico `SHOP_TOPIC_EVENT` em Python. Essa aplicação disponibilizará algumas rotas para mostrar as as últimas compras feitas em nossa loja.

12.1 Configurando a aplicação

Para configurar uma aplicação Python, basta criar um arquivo chamado `requirements.txt` na pasta raiz do projeto e adicionar o nome de todas as bibliotecas que serão utilizadas. Pode ser adicionada também a versão da biblioteca, mas, como utilizaremos a versão mais recente no nosso projeto, não precisamos adicionar a versão.

Na nossa aplicação, vamos utilizar duas bibliotecas, o Flask, que é um framework para o desenvolvimento de serviços REST, e o `kafka-`

`python`, que é a biblioteca que implementa o cliente para conexão com o Kafka.

```
Flask
kafka-python
```

Para instalar as dependências definidas no arquivo da máquina local, basta executar o comando `pip install -r requirements.txt`. Para executar esse comando, o Python já deve ter sido instalado na máquina. Na primeira edição do livro, a aplicação foi executada na versão 3.5 do Python; na versão atual foi utilizado o Python 3.11 e não foi necessária nenhuma mudança no código. Então, qualquer versão acima da 3.5 deve executar a aplicação sem problemas.

Para organizar o projeto, eu criei uma pasta chamada `src` na qual ficará todo o código-fonte da aplicação. Nessa pasta, eu criei dois arquivos Python, o primeiro, o `consumer.py`, que terá a implementação da classe que consumirá as mensagens do Kafka, e o arquivo `main.py`, que terá a inicialização do consumidor e a implementação das rotas com o Flask.

12.2 Implementando o cliente do Kafka

O primeiro passo na implementação do cliente do Kafka é criar uma classe que será o consumidor do Kafka. Vamos implementar a classe `Consumer`, e ela será uma classe filha da `threading.Thread` pois o consumidor do Kafka será executado em uma thread separada. Precisaremos de duas threads na nossa aplicação, já que uma executará o consumidor do Kafka e a outra a API REST implementada com o Flask.

```
import threading
import json
import logging
from kafka import KafkaConsumer
```

```
class Consumer(threading.Thread):

    def __init__(self):
        threading.Thread.__init__(self)
        self.stop_event = threading.Event()

    # outros métodos da classe
```

Com a classe definida, podemos implementar o método `run` dentro da nossa classe. Como essa classe será uma thread, a implementação do método `run` é obrigatória. Nesse método, primeiramente iniciaremos a conexão com o Kafka, deixando o consumidor pronto para receber as mensagens utilizando a classe `KafkaConsumer` que pertence à biblioteca `kafka-python`.

Note que na chamada à classe `KafkaConsumer` definimos o servidor que estamos conectando (`localhost:9092`), o identificador do grupo desse consumidor (`grupo_python`) e o timeout da conexão (30000 milissegundos ou 30 segundos). Depois inscrevemos o nosso consumidor ao tópico `SHOP_TOPIC_EVENT` chamando o método `subscribe`.

Depois, criamos um loop que vai ser executado enquanto o consumidor estiver funcionando. Esse loop fica esperando uma mensagem ser recebida no tópico e, quando ela é recebida, a mensagem é processada pelo o código que está dentro do `for`. Esse código escreve uma mensagem no console indicando que uma mensagem foi recebida e a adiciona a uma lista, que é a lista que depois será retornada para o usuário nas rotas do Flask.

```
def run(self):
    try:
        consumer = KafkaConsumer(
            bootstrap_servers=['localhost:9092'],
            group_id="grupo_python",
            consumer_timeout_ms=30000
        )
```

```
            consumer.subscribe(['SHOP_TOPIC_EVENT'])

            while not self.stop_event.is_set():
                for message in consumer:
                    logging.info(
                        'Receiving message from topic SHOP_TOPIC_EVENT'
                    )
                    self.messages.append(
                        json.loads(message.value.decode("utf-8"))
                    )

                consumer.close()
            except IOError as exception:
                print("Read error")
                logging.error(exception)

    def get_messages(self):
        return self.messages

    messages = []
```

Além disso, temos também o método `get_messages`, que apenas retorna a lista com todas as mensagens que já foram lidas no Kafka.

12.3 Inicializando o consumidor

No código anterior, criamos a classe que implementa o consumidor do Kafka, mas ainda não iniciamos a execução dele. Para isso, agora no arquivo `main.py` devemos criar um objeto do tipo `Consumer` e chamar o seu método `start`.

A chamada desse método chamará o método `run`, que implementamos na classe `Consumer`, mas fará isso em uma thread separada, o que permitirá que a nossa aplicação execute tanto o consumidor do Kafka quanto as rotas REST.

```
from flask import Flask
from consumer import Consumer
import json
import logging

# inicializa o consumidor
consumer = Consumer()
consumer.start()
```

Depois da execução do método `start`, o consumidor será executado e, sempre que uma mensagem for adicionada ao tópico `SHOP_TOPIC_EVENT`, ela será armazenada em uma lista que terá todas as mensagens recebidas.

Essas mensagens poderão ser visualizadas pelos usuários utilizando as rotas que desenvolveremos a seguir.

12.4 Implementando uma rota com o Flask

Agora que já estamos lendo as mensagens no Kafka, podemos implementar algumas rotas para retornar essas mensagens para os usuários. Faremos isso no mesmo arquivo `main.py`. Existem diversos frameworks Python que podem ser utilizados para isso, sendo que o Flask é um dos mais conhecidos e um dos mais fáceis de usar. Para utilizar o Flask, apenas temos que criar um objeto Flask com o comando `Flask(__name__)`.

Com isso, apenas colocamos o decorator `@app.route('/')` em um método, e ele já é exposto como uma rota REST em nossa aplicação. Criamos três rotas em nossa aplicação, uma que retornará todas as mensagens que foram recebidas, a `@app.route('/all')`, uma que retornará apenas a última mensagem recebida, a

`@app.route('/last')`, e uma que retornará as últimas dez mensagens recebidas no tópico, a `@app.route('/last_ten')`.

```
@app.route('/all')
def get_messages():
    messages = consumer.get_messages()
    return json.dumps(messages)

@app.route('/last')
def get_last():
    messages = consumer.get_messages()

    if len(messages) > 0:
        return json.dumps(messages[-1])
    else:
        return 'lista vazia'

@app.route('/last_ten')
def get_last_ten():
    messages = consumer.get_messages()

    if len(messages) > 9:
        return json.dumps(messages[-10:])
    else:
        return json.dumps(messages[-len(messages):])

if __name__ == '__main__':
     app.run()
```

Note que o primeiro método, o que implementa a rota que retorna todas as mensagens, é mais simples. Ele sempre retorna a lista de mensagens completa e, se a lista for vazia, apenas retorna a lista vazia.

Porém, os outros dois métodos, que implementam as outras rotas, são um pouco mais complexos. Como estamos lidando com os índices da lista, no primeiro método precisamos verificar se existe pelo menos um item, e no segundo verificar qual o tamanho da lista, pois queremos retornar as últimas dez compras, mas pode ser que

existam menos compras do que isso na lista. Se não fizermos esse tratamento, pode dar um erro de índice da lista inválida quando chamarmos as rotas.

> **DECORATOR**
>
> Um decorator em Python é como uma anotação em Java, ele permite adicionar uma funcionalidade para um método sem ter que alterar esse método. Com o Flask, utilizamos o decorator `@app.route` para criar as rotas REST em nossa aplicação.

12.5 Executando a aplicação

Agora podemos executar a aplicação. Para isso, devemos executar o comando `python3 main.py` dentro da pasta `scr` do projeto. Com esse comando, será iniciado o consumidor do Kafka e também estarão disponíveis as rotas do Flask. No console, devem aparecer algumas mensagens indicando que o servidor foi iniciado. Uma informação importante nesse log indica em qual porta o servidor foi inicializado. A porta padrão do Flask é a 5000 e, para acessar as rotas, devemos chamar a URL `http://localhost:5000` .

```
eduardo@eduardo:~/livro/livro-kafka/python-consumer/src$ python3 main.py
 * Serving Flask app 'main' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production
deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Agora que a aplicação foi inicializada, podemos chamar a rota `GET /all` , que retornará todas as mensagens que foram recebidas no Kafka desde o início da execução da aplicação, como no exemplo:

```
[
    {
        "identifier": "4e8345df-62b3-4030-8760-1b201722267e",
        "status": "SUCCESS",
        "buyerIdentifier": "b-3",
        "items": [
            {
                "productIdentifier": "123456789",
                "amount": 1,
                "price": 1000.0
            }
        ]
    },
    {
        "identifier": "a146b7b5-6d2a-4ea2-a0a2-a5b7c07f89af",
        "status": "SUCCESS",
        "buyerIdentifier": "b-3",
        "items": [
            {
                "productIdentifier": "123456789",
                "amount": 1,
                "price": 1000.0
            }
        ]
    },
    {
        "identifier": "c91d4f77-460c-4ccf-9930-7df0af5ebecd",
        "status": "SUCCESS",
        "buyerIdentifier": "b-3",
        "items": [
            {
                "productIdentifier": "123456789",
                "amount": 1,
                "price": 1000.0
            }
        ]
    }
]
```

Podemos também chamar a rota `GET /last`, que retornará apenas a última mensagem recebida do Kafka.

```
{
    "identifier": "c91d4f77-460c-4ccf-9930-7df0af5ebecd",
    "status": "SUCCESS",
    "buyerIdentifier": "b-3",
    "items": [
        {
            "productIdentifier": "123456789",
            "amount": 1,
            "price": 1000.0
        }
    ]
}
```

Agora temos mais um serviço rodando em nossa aplicação, e o mais interessante é que essa aplicação foi desenvolvida em uma linguagem diferente, o Python. Porém, isso não é um problema para o Kafka, pois as mensagens podem ser enviadas no formato `JSON`, que é independente de linguagem, por isso podemos produzir e consumir mensagens em um sistema desenvolvido em qualquer linguagem que possua um cliente do Kafka disponível.

Por enquanto, utilizamos as configurações padrão na execução do Kafka, do consumidor e do produtor. Porém, o Kafka possui diversas configurações que podem facilitar a implementação e até mesmo melhorar o desempenho da aplicação. No próximo capítulo, veremos alguns exemplos dessas configurações e como alterá-las.

Capítulo 13
Configurações do Kafka

O Kafka possui várias configurações que podem ser feitas para alterar o comportamento da ferramenta ou para melhorar o desempenho da aplicação dependendo do caso de uso. Por exemplo, já alteramos uma configuração do Kafka para excluir os tópicos no capítulo 8, pois, por padrão e por segurança, a ferramenta impede a exclusão de tópicos. Porém o Kafka possui centenas de outras configurações que podemos alterar.

Neste capítulo, veremos três tipos de configurações, as que podem ser alteradas diretamente na ferramenta e que valerão para todo o Kafka, as que podem ser definidas para cada um dos consumidores e as que podem ser alteradas somente para um produtor específico.

As configurações do Kafka são definidas nos arquivos `server.properties` e `zookeper.properties`, que estão na instalação do Kafka que configuramos no capítulo 2. As configurações dos consumidores e produtores serão feitas diretamente no código das aplicações, na classe `KafkaConfig` que desenvolvemos no capítulo 4.

13.1 Configurações gerais

As configurações gerais do Kafka são definidas em dois arquivos, o `zookeper.properties` e o `server.properties`, que estão dentro do diretório `config` da instalação do Kafka. Ambos possuem configurações do Kafka, como onde serão armazenados os arquivos dos tópicos, em qual porta as ferramentas serão executadas e se o Kafka permite a exclusão de tópicos ou não. Vamos ver aqui algumas das principais configurações que podemos fazer nesses arquivos, começando com o `zookeper.properties`.

A primeira configuração é `clientPort`, que indica a porta em que o Zookeeper vai rodar. Por padrão, é a porta 2181, mas não há muitos motivos para trocá-la. A segunda é se queremos habilitar a tela de administração do Zookeeper, a `admin.enableServer`. Por padrão, essa configuração vem desabilitada, mas se por algum motivo for necessária uma administração mais forte do Kafka e do Zookeeper, podemos habilitar essa configuração. Caso habilitemos a tela de administração do Zookeeper, podemos alterar a porta dessa tela na configuração `admin.serverPort`.

```
clientPort=2181
admin.enableServer=false
# admin.serverPort=8080
```

Obviamente existem diversas outras configurações no Zookeeper, mas essas são as principais para um usuário do Kafka. Vamos ver as configurações definidas no arquivo `server.properties`. Um exemplo de configuração é definir o identificador do servidor. Esse dado não afetará a execução da aplicação, mas pode ser importante quando for necessário debugar o funcionamento do Kafka.

```
broker.id=0
```

Uma configuração bastante importante é o número padrão de partições. Vimos nos capítulos anteriores que quando criamos um tópico, por padrão, ele terá apenas uma partição. Podemos mudar esse número padrão com a propriedade `num.partitions`, assim os novos tópicos criados com as propriedades padrão terão esse número de partições.

```
num.partitions=1
```

Existem diversas propriedades que definem como os arquivos com as mensagens serão salvos. Por exemplo, com a propriedade `log.dirs`, é possível alterar o diretório onde os arquivos serão salvos. No Linux, por padrão, os arquivos são salvos no diretório `/tmp/kafka-logs`, o que em algumas distribuições, como o Ubuntu, faz com que os arquivos sejam deletados sempre que o sistema é

reiniciado. Por isso, caso não queira que os dados sejam excluídos, sempre mude esse diretório.

```
log.dirs=/tmp/kafka-logs
```

A propriedade `log.retention.hours` indica por quantas horas os arquivos estarão disponíveis nos tópicos. O número padrão é 168, que são 7 dias, mas podemos aumentar ou diminuir esse tempo dependendo da nossa aplicação. Além disso, podemos definir também o tamanho máximos dos arquivos, assim, se o tamanho total dos arquivos passar de um determinado tamanho, as mensagens começarão a ser apagadas independente do tempo em que eles ficaram armazenadas.

```
log.dirs=/tmp/kafka-logs
log.retention.hours=168
log.retention.bytes=1073741824
```

Também podemos configurar em qual instância do Zookeeper queremos que o Kafka conecte. Por padrão, ele tenta se conectar no `localhost:2181`, mas podemos trocar essa configuração caso o Zookeeper esteja sendo executado em outra máquina ou com uma porta diferente.

```
zookeeper.connect=localhost:2181
zookeeper.connection.timeout.ms=18000
```

Outra configuração importante, que já vimos no capítulo 8, é para habilitar a deleção de tópicos, que é desabilitado por padrão, mas podemos mudar isso caso seja necessário recriar um tópico.

```
delete.topic.enable=true
```

13.2 Configurações do consumidor

Além das configurações gerais do Kafka, também existem diversas configurações que podem ser alteradas para cada um dos consumidores. Essas configurações podem ser alteradas diretamente no código do Spring Boot. Podemos fazer isso na classe `KafkaConfig`, que criamos no capítulo 4, no método `consumerFactory`. Nesse método, criamos um mapa chamado `props`, no qual podemos adicionar diversas configurações com seus respectivos valores. No código do capítulo 4, usamos apenas uma configuração que foi a `BOOTSTRAP_SERVERS_CONFIG`, passando a URL de conexão do Kafka, como podemos ver no código a seguir.

```
public ConsumerFactory<String, ShopDTO> consumerFactory() {
    JsonDeserializer<ShopDTO> deserializer =
        new JsonDeserializer<>(ShopDTO.class);

    Map<String, Object> props = new HashMap<>();
    props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG
        , bootstrapAddress);

    return new DefaultKafkaConsumerFactory<>(
        props,
        new StringDeserializer(),
        deserializer);
}
```

Outra configuração que já está definida no código anterior são os *deserializers*, que são passados no construtor da classe `DefaultKafkaConsumerFactory`. Note que temos dois deserializers: o primeiro é o `StringDeserializer`, pois as chaves das mensagens do Kafka serão do tipo String, e o segundo é um `JsonDeserializer`, que recebe como parâmetro a nossa classe `ShopDTO`, que indica que as mensagens do Kafka serão no formato JSON, construídas com os atributos da classe `ShopDTO`.

O processo de serialização/deserialização é bastante importante no Kafka. O Spring Boot faz esse processo ser transparente para o programador, mas é importante entender o que isso significa. O box a seguir explica o processo de serialização/deserialização.

SERIALIZAÇÃO/DESERIALIZAÇÃO DOS DADOS

Para quem não conhece esse termo, serializar os dados significa converter os dados para um formato que facilite a troca de dados entre as aplicações. Um exemplo é converter os dados de um objeto para JSON, ou então para bytes ou para XML. O processo de deserialização é o contrário, é converter o dado para o seu formato original, do JSON para o objeto Java, por exemplo. O Kafka faz todo o processo de serialização/deserialização para a pessoa programadora, só é necessário indicar quais formatos serão utilizados.

Além das configurações que já utilizamos, podemos adicionar diversas outras. Para isso, basta adicionar mais valores ao objeto `props` que foi definido no código mostrado anteriormente. Vamos ver algumas configurações interessantes que podemos utilizar.

Criação automática de tópicos

A primeira configuração é a `ALLOW_AUTO_CREATE_TOPICS_CONFIG`, que indica se um tópico pode ser automaticamente criado ou não. Vimos anteriormente que, quando criamos um consumidor para um tópico que não existe, esse tópico é criado automaticamente quando ele é utilizado pela primeira vez. Podemos alterar esse comportamento colocando um valor `false` para essa propriedade. O valor `true` é o padrão para essa configuração.

```
props.put(ConsumerConfig.ALLOW_AUTO_CREATE_TOPICS_CONFIG, false);
```

Nesse caso, se tentarmos utilizar um tópico que não existe, a aplicação retornará um erro.

Quantidade de registros

Quando um consumidor se conecta ao Kafka para ler as mensagens em uma partição de um tópico do Kafka, ele pode já ler diversas

mensagens de uma só vez. Isso evita que o consumidor tenha que ficar se conectando toda hora no Kafka e aumenta o desempenho da aplicação. Podemos alterar esse comportamento com a configuração `MAX_POLL_RECORDS_CONFIG`, que indica qual o máximo de mensagens que um consumidor pode ler de uma vez só.

Por padrão, esse valor é 500, mas podemos aumentar ou diminuir esse valor dependendo da necessidade da nossa aplicação. Essa alteração apenas muda o comportamento interno do consumidor, evitando as diversas conexões ao Kafka. A implementação do consumidor não precisa de nenhuma mudança, pois as mensagens serão processadas na ordem em que forem coletadas no Kafka. O código a seguir mostra a mudança nessa configuração.

```
props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 1000);
```

Leitura de registros

É possível também configurar desde quando os consumidores começarão a ler os dados dos tópicos, se desde o início do tópico, ou só a partir das mensagens que chegaram depois que o consumidor se conectou ao tópico.

O valor padrão é o `latest`, que indica que serão lidas apenas as novas mensagens, isto é, as enviadas após o consumidor se conectar ao tópico. Porém, é possível mudar esse valor para `earliest`, que indica que as mensagens serão lidas desde a primeira mensagem que está no tópico. Para mudar isso, podemos usar a configuração `AUTO_OFFSET_RESET_CONFIG`. O código a seguir mostra como mudar o valor para essa configuração.

```
props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
```

Definição do Consumer Group

Podemos configurar toda a aplicação para usar o mesmo `Consumer Group` usando a configuração `GROUP_ID_CONFIG`. Assim, todos os consumidores implementados na aplicação estarão automaticamente

no mesmo grupo. Com essa configuração, não precisamos definir o grupo na anotação `@KafkaListener` como fizemos nos capítulos anteriores.

```
props.put(ConsumerConfig.GROUP_ID_CONFIG, "consumer-group");
```

Intervalo de notificações

O Kafka usa um mecanismo chamado `heartbeat` para que o consumidor notifique o Kafka que ele ainda está funcionando. Podemos configurar o intervalo para essas mensagens com a configuração `HEARTBEAT_INTERVAL_MS_CONFIG`. Por padrão, esse intervalo é de 3.000 milissegundos, mas podemos alterá-lo para outro intervalo, sempre em milissegundo, como no exemplo a seguir em que o intervalo é alterado para mil milissegundos.

```
props.put(ConsumerConfig.HEARTBEAT_INTERVAL_MS_CONFIG, 1000);
```

Timeout das sessões

Também relacionado aos `heartbeats`, podemos configurar um `timeout` para o Kafka considerar que o consumidor não está mais conectado. Com a `SESSION_TIMEOUT_MS_CONFIG` é definido um tempo máximo pelo qual um consumidor pode ficar sem mandar um `heartbeat`. Depois desse tempo, o Kafka considerará que o consumidor não está mais funcionando e rebalanceará as partições entre os consumidores restantes.

```
props.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, 10000);
```

Commits automáticos

Outra configuração bastante importante é habilitar ou não os commits automáticos. Por padrão, quando um consumidor inicia o processamento de uma mensagem, ele já avisa o Kafka que essa mensagem foi lida, assim o Kafka sabe que aquela mensagem já foi processada. Porém, em algumas aplicações, só vamos querer que

seja confirmado que a mensagem foi lida quando nenhum erro ocorre.

Podemos mudar esse comportamento padrão utilizando a configuração `ENABLE_AUTO_COMMIT_CONFIG`. Com isso, poderemos mudar o momento do `commit` da mensagem, para que ele não seja feito logo no início, mas apenas no fim do processamento da mensagem. Isso permite que façamos retentativas caso o processamento de uma mensagem dê algum problema.

```
props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
```

Aqui listei algumas das configurações que considero mais interessantes e as que podem influenciar o desempenho da aplicação. Existem outras configurações relacionadas ao uso de `SSL` para autenticação no Kafka e também para o cálculo de métricas no Kafka. Na documentação do Kafka, existe uma página com todas as configurações disponíveis para os consumidores.

13.3 Configurações do produtor

Da mesma forma que podemos mudar algumas configurações para cada consumidor, podemos alterá-las para os produtores. Os produtores normalmente são mais simples que os consumidores, ainda assim temos configurações interessantes que podem melhorar a implementação da aplicação.

Também já definimos algumas configurações para os produtores no capítulo 4, como podemos ver no código a seguir:

```
public ProducerFactory<String, ShopDTO> producerFactory() {
    Map<String, Object> props = new HashMap<>();
    props.put(
        ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
```

```
            bootstrapAddress);
        props.put(
            ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            StringSerializer.class);
        props.put(
            ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            JsonSerializer.class);

        return new DefaultKafkaProducerFactory<>(props);
    }
```

Nesse código, definimos o `BOOTSTRAP_SERVERS_CONFIG`, que indica o endereço do Kafka, e as classes `KEY_SERIALIZER_CLASS_CONFIG` e `VALUE_SERIALIZER_CLASS_CONFIG`, que farão a serialização dos dados que serão enviados para o Kafka. Note que, assim como no consumidor, aqui também utilizamos a `StringSerializer` e a `JsonSerializer`. A diferença é que no produtor fazemos a serialização dos dados, e no consumidor fazemos a deserialização dos dados.

Tamanho das mensagens

Uma configuração interessante para o caso de o produtor enviar muitas mensagens em sequência é a `BATCH_SIZE_CONFIG`. Ela define um tamanho em bytes para acumular diversas mensagens para serem enviadas de uma só vez em apenas uma conexão com o Kafka. Isso pode melhorar a performance da aplicação, pois diminui o número de conexões necessárias com o Kafka.

O valor dessa configuração é definido em `bytes` e não no número de mensagens. O valor padrão é 16384 bytes e, caso o valor passado na configuração seja 0, o batch é desabilitado totalmente. Para cada mensagem enviada pelo produtor, uma nova conexão será feita.

```
props.put(ProducerConfig.BATCH_SIZE_CONFIG, "16384");
```

Identificador do produtor

Para facilitar a rastreabilidade das mensagens é possível definir um identificador para os produtores com a configuração `CLIENT_ID_CONFIG`. Essa configuração não tem nenhum efeito prático, pois o Kafka processará a mensagem da mesma forma se ela tiver esse identificador ou não. A utilidade dessa configuração é apenas para facilitar a depuração da aplicação e saber qual a origem da mensagem.

Ela pode ser bastante útil em casos em que diversos produtores enviam mensagens para um mesmo tópico. Por padrão, esse valor é vazio, sendo impossível rastrear a origem da mensagem caso existam mais de um produtor.

```
props.put(ProducerConfig.CLIENT_ID_CONFIG, "client-1");
```

Timeout do produtor

É possível também configurar o máximo de tempo pelo qual uma conexão de um produtor pode ficar sem utilização, isto é, sem enviar mensagens para o Kafka, com a configuração `CONNECTIONS_MAX_IDLE_MS_CONFIG`. Por padrão, esse valor é 54.000 milissegundos (9 minutos), então, se sabemos de antemão que nossa aplicação vai ter intervalos muito maiores que esse para o envio da mensagem, vale a pena diminuir esse valor para que a conexão com o Kafka seja liberada mais rapidamente.

```
props.put(ProducerConfig.CONNECTIONS_MAX_IDLE_MS_CONFIG, 10000);
```

As configurações são importantes para melhorar o desempenho das aplicações, como as que enviam mensagens em `batch` no produtor, ou para facilitar a implementação de alguma funcionalidade como a que possibilita retentativas caso o processamento das mensagens falhem. Como dito anteriormente, existem centenas de possíveis configurações no Kafka. Eu recomendo ao leitor ou leitora que verifique a documentação do Kafka para que veja todas as possíveis configurações. Lá existe uma lista com todas as configurações possíveis, todos os valores aceitos para elas e os valores padrões.

Agora nossas aplicações estão prontas e configuradas. O problema agora é que executar diversas aplicações de uma única vez ficou um pouco complexo. No próximo capítulo, veremos como utilizar o Docker para facilitar um pouco as coisas. Primeiro, vamos ver como criar os contêineres Docker com as nossas aplicações e depois, como executar também o Kafka com o Docker. Isso facilitará demais tanto a configuração da máquina quanto a execução das aplicações.

Capítulo 14
Executando todas as aplicações com o Docker

O Docker facilita muito a execução das aplicações sem que tenhamos que configurar tudo toda vez que tivermos que rodá-las em máquinas diferentes. Com ele, podemos criar imagens das nossas aplicações, que depois podem ser executadas em qualquer máquina de forma bastante simples. Além disso, utilizando o *docker-compose* podemos subir um conjunto de aplicações utilizando apenas um comando. Por isso, neste capítulo, veremos como construir as imagens das aplicações com o Docker e depois como executar todas as aplicações desenvolvidas até aqui utilizando o docker-compose.

A primeira coisa que teremos que fazer é criar o `Dockerfile` para todas as aplicações. Esse arquivo tem todas as configurações para a criação dos contêineres para as aplicações, depois disso mostraremos como criar cada uma das imagens do Docker para as aplicações e, por fim, utilizaremos o docker-compose que é uma forma de simplificar a execução de diversos contêineres utilizando apenas um comando.

14.1 Mudanças nas aplicações

Teremos que fazer algumas pequenas alterações nas nossas aplicações para executá-las com o Docker. A primeira mudança é adicionar o plugin `spring-boot-maven-plugin` aos arquivos `pom.xml`. Ele serve para construir o arquivo `jar` da aplicação corretamente, de modo que é possível executar a aplicação apenas utilizando `java -jar arquivo.jar`, sendo o `arquivo.jar` o nome do arquivo que foi gerado pelo comando `mvn clean install`. Sem esse plugin, a execução do `jar` não funcionará. As seguintes linhas devem ser

adicionadas ao arquivo `pom.xml` das aplicações `shop-api`, `shop-validator` e `shop-report`.

Além disso, precisamos adicionar um plugin para que o Maven considere as anotações do Lombok para gerar os códigos necessários. Esse plugin é o `maven-compiler-plugin`, com a tag `annotationProcessorPaths`, que indica que, no momento em que o projeto é compilado, as anotações do Lombok devem ser processadas. O trecho a seguir mostra o código XML que deve ser adicionado no arquivo `pom.xml` em todos os projetos Java.

```
<build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
      <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>${maven-compiler-plugin.version}</version>
          <configuration>
              <annotationProcessorPaths>
                  <path>
                      <groupId>org.projectlombok</groupId>
                      <artifactId>lombok</artifactId>
                      <version>${lombok.version}</version>
                  </path>
              </annotationProcessorPaths>
          </configuration>
      </plugin>
        </plugins>
    </build>
```

Além disso, precisaremos adicionar ao arquivo `application.properties` das três aplicações o endereço do Kafka que será utilizado nos contêineres do Docker, por meio da configuração `kafka.bootstrapAddress`. O código a seguir mostra como deve ser feita a definição dessa propriedade.

```
kafka.bootstrapAddress=kafka:9092
```

O endereço `kafka:9092` será definido quando subirmos nossas aplicações no Docker. Vamos criar uma rede com todos os contêineres, e esse endereço será criado dentro dessa rede.

Antes, essa configuração no arquivo `application.properties` não era necessária, pois estávamos utilizando o valor padrão que está definido no código, que é o `localhost:9092`. Para relembrar, o código a seguir mostra onde essa configuração está sendo utilizada. Na anotação `@Value`, note que está sendo utilizado o valor da propriedade `kafka.bootstrapAddres`, mas como ela não estava definida anteriormente, estava sendo utilizado o valor padrão que foi definido depois dos dois-pontos.

```
@Configuration
public class KafkaConfig {

    @Value(value = "${kafka.bootstrapAddress:localhost:9092}")
    private String bootstrapAddress;

    ...
```

Também temos que fazer uma pequena mudança na aplicação Python para que ele aponte para esse novo endereço do Kafka, assim, teremos que alterar o endereço do Kafka na classe `Consumer` para o endereço `kafka:9092`. Lembre-se de que antes essa configuração estava também apontando para o endereço `localhost:9092`.

```
consumer = KafkaConsumer(
    bootstrap_servers=['kafka:9092'],
    group_id="grupo_python",
    consumer_timeout_ms=30000
)
```

14.2 Criando os contêineres

Depois das alterações nas aplicações, temos que construir os contêineres. Para isso, primeiro criaremos o `Dockerfile` para todos os projetos. Para as aplicações Java, esses arquivos serão bastante parecidos, pois serão construídos a partir da imagem `openjdk:11-jdk`, que é um Linux com o Java 11 já instalado. Depois, copiaremos o arquivo `jar` para o contêiner e executaremos a aplicação com o comando `java -jar app.jar`. O exemplo a seguir mostra o `Dockerfile` da `shop-api`.

```
FROM openjdk:21-jdk

VOLUME /tmp
ARG JAR_FILE=target/shop-api-1.0-SNAPSHOT.jar
COPY ${JAR_FILE} app.jar
EXPOSE 8080

ENTRYPOINT ["java","-jar","/app.jar"]
```

Um arquivo `Dockerfile` possui diversos comandos, e explicar todos foge do escopo deste livro, mas os que utilizamos nesses exemplos foram:

- `FROM`: Indica um contêiner base a partir do qual construiremos o nosso contêiner. Como precisamos da JDK, criaremos nosso contêiner a partir de um contêiner Linux que já possui a JDK 21 instalada.
- `VOLUME`: Cria uma pasta caso ela não exista, e depois todos os comandos executados no Dockerfile serão executados dentro dessa pasta.
- `ARG`: Define uma variável que pode ser utilizada no Dockerfile. No nosso caso, definimos o caminho para a aplicação na variável `JAR_FILE`.
- `COPY`: Copia um arquivo da máquina local para o contêiner, no caso, copiamos o arquivo `jar` para o contêiner, já o renomeando para `app.jar`.

- `EXPOSE` : Indica uma porta de entrada para o contêiner, por exemplo, na `shop-api` a porta de entrada será a 8080, então vamos expor a porta 8080 do contêiner.
- `ENTRYPOINT` : Define o comando que será executado quando o contêiner for iniciado. No nosso exemplo, executaremos o comando `java -jar` para iniciar a aplicação.

Depois, para construir a imagem desse projeto, podemos executar o comando `docker build -t shop-api .`. O `-t` indica qual é o nome da imagem que será construída, e não esqueça do ponto final ao fim do comando, que indica para o Docker buscar o arquivo Dockerfile no diretório corrente. Lembre também de executar esse comando dentro da pasta onde está o projeto `shop-api`.

A criação das próximas imagens será bem parecida, pois toda aplicação Spring Boot possui a mesma estrutura, apenas precisamos ter a JDK instalada e copiar o arquivo `jar` para o contêiner. O código a seguir mostra o `Dockerfile` para a aplicação `shop-report`.

```
FROM openjdk:21-jdk

VOLUME /tmp
ARG JAR_FILE=target/shop-report-1.0-SNAPSHOT.jar
COPY ${JAR_FILE} app.jar
EXPOSE 8082

ENTRYPOINT ["java","-jar","/app.jar"]
```

Para construir a imagem desse projeto, podemos executar o comando `docker build -t shop-report .` dentro do diretório da aplicação `shop-report`.

Também seguindo os mesmos passos das aplicações anteriores, o código a seguir mostra o `Dockefile` para a aplicação `shop-validator`.

```
FROM openjdk:21-jdk

VOLUME /tmp
ARG JAR_FILE=target/shop-validator-1.0-SNAPSHOT.jar
```

```
COPY ${JAR_FILE} app.jar
EXPOSE 8081

ENTRYPOINT ["java","-jar","/app.jar"]
```

Para construir a imagem desse projeto, podemos executar o comando `docker build -t shop-validator .` dentro do diretório da aplicação `shop-validator`.

Também seguindo os mesmos passos das aplicações anteriores, o código a seguir mostra o `Dockefile` para a aplicação `shop-retry`.

```
FROM openjdk:21-jdk

VOLUME /tmp
ARG JAR_FILE=target/shop-retry-1.0-SNAPSHOT.jar
COPY ${JAR_FILE} app.jar

ENTRYPOINT ["java","-jar","/app.jar"]
```

Para construir a imagem desse projeto, podemos executar o comando `docker build -t shop-retry .` dentro do diretório da aplicação `shop-retry`.

Assim como para as aplicações Java, construir a imagem Docker de um projeto Python também é bastante simples. Será necessário também desenvolvermos um arquivo `Dockerfile` para esse projeto e os comandos são praticamente os mesmos, o único comando novo é o `RUN`, que simplesmente executa um comando no sistema operacional do contêiner, por exemplo, o `pip install` e o `cd src`.

```
FROM python:3.8

WORKDIR /code

COPY . .
RUN pip install -r requirements.txt
EXPOSE 5000
```

```
RUN cd src

CMD [ "python3", "-m" , "flask", "run", "--host=0.0.0.0"]
```

Para construir a imagem desse projeto, podemos executar o comando `docker build -t shop-python .` dentro do diretório da aplicação Python.

Podemos verificar se todas as imagens foram construídas corretamente usando o comando `docker images`. Se tudo funcionou corretamente, o retorno desse comando deve ser:

```
eduardo@eduardo:~/livro/livro-kafka/python-consumer$ docker images
REPOSITORY                      TAG          IMAGE ID       CREATED
SIZE
shop-python                     latest       627b991399c5   3 minutes ago
922MB
shop-validator                  latest       0cfb81a1da7a   8 days ago
712MB
shop-report                     latest       22234cf04bed   8 days ago
712MB
shop-api                        latest       f9b4e16cc9b5   8 days ago
712MB
shop-retry                      latest       543444df5be1   8 days ago
712MB
```

Note que esse comando lista todas as imagens que existem na sua máquina, mostrando quando elas foram criadas e o tamanho da imagem em disco. O Docker também sempre cria um identificador único para a imagem e a tag define a versão da imagem. Como não definimos uma versão específica no comando `docker build`, as imagens recebem a tag padrão `latest`.

14.3 docker-compose

Agora que temos todas as imagens construídas, podemos utilizar o `docker-compose` para executar todas as aplicações e também o Kafka utilizando o Docker. Com ele, teremos que definir em um arquivo `yaml` com um conjunto de contêineres que serão executados. Depois que esse arquivo for definido, podemos executar todos os contêineres apenas utilizando o comando `docker-compose up`. Vamos ver passo a passo como construir esse arquivo e depois como executar a aplicação.

Inicializando o YAML

Todos os arquivos do `docker-compose` devem ter duas linhas inicias, uma que define a versão do formato do arquivo `docker-compose`, no nosso caso, vamos utilizar a versão 3.6, e depois outra com a declaração de uma lista de serviços, o que é feito com a palavra `services`. A partir dessa linha serão definidos os contêineres que serão executados.

```
version: "3.6"
services:
```

Configurando o Kafka e o Zookeeper

Agora definiremos os `services` que serão executados. Vamos começar com o serviço do Kafka. Existem várias imagens prontas para executar o Kafka, recomendo usar a imagem oficial chamada `apache/kafka`. Por isso, não precisamos criar os `Dockerfile` para o Kafka, podemos apenas utilizar a imagem já disponível.

Para definir o serviço do Kafka, vamos criar um `service` chamado `kafka`. Para ele, precisaremos definir apenas três coisas: qual imagem Docker vamos utilizar, qual a porta de acesso ao serviço, e algumas variáveis de ambiente que serão explicadas mais para a frente.

Definimos a imagem na linha `image`, que receberá o valor `apache/kafka:latest`. Essa imagem ainda não existe na nossa máquina, mas veremos que, quando formos executar a aplicação, o Docker fará o download dessa imagem automaticamente.

Além disso, na configuração `ports`, temos que fazer a configuração da porta que a aplicação será acessada (no caso, a 9092) e, na configuração `environment`, temos que definir as variáveis de ambiente, que são:

- `KAFKA_NODE_ID`: Define o id do Kafka em um cluster. É obrigatório sempre definir um id para o cluster.
- `KAFKA_PROCESS_ROLES`: Define os papeis da instância em um cluster Kafka. O `broker` indica que o nó será responsável por armazenar e processar mensagens, e o `controller` indica que o nó também gerenciará a configuração do cluster.
- `KAFKA_LISTENERS`: Define uma lista de endereços nos quais o Kafka está disponível para ser acessado. O primeiro é o endereço que será acessado pelas aplicações `PLAINTEXT://kafka:29092`, e o segundo, o `CONTROLLER://localhost:9093`, será utilizado internamente pelo Kafka.
- `KAFKA_ADVERTISED_LISTENERS`: Define os endereços externos do cluster do Kafka. Esse é o endereço que usaremos em nossas aplicações para acessar o cluster do Kafka.
- `KAFKA_CONTROLLER_LISTENER_NAMES`: Define qual listener será usado pelos controladores para comunicação no cluster. Apenas para uso interno do Kafka.
- `KAFKA_LISTENER_SECURITY_PROTOCOL_MAP`: Configura qual será o protocolo de autenticação nas comunicações internas e externas do Kafka. No caso, o `PLAINTEXT` indica que não haverá nenhum mecanismo de segurança. Para quem precisar de segurança na comunicação com o Kafka, é possível utilizar também `SSL` para autenticação.
- `KAFKA_CONTROLLER_QUORUM_VOTERS`: Configura os controladores no cluster Kafka.

- `KAFKA_NUM_PARTITIONS` : Essa variável não é obrigatória, mas indica que o Kafka deve criar os tópicos com 3 partições por padrão.

A definição completa do serviço do Kafka será:

```
  kafka:
    image: apache/kafka:latest
    environment:
      KAFKA_NODE_ID: 1
      KAFKA_PROCESS_ROLES: broker,controller
      KAFKA_LISTENERS: PLAINTEXT://kafka:9092,CONTROLLER://localhost:9093
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka:9092
      KAFKA_CONTROLLER_LISTENER_NAMES: CONTROLLER
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT
      KAFKA_CONTROLLER_QUORUM_VOTERS: 1@localhost:9093
      KAFKA_NUM_PARTITIONS: 3
```

Quando rodamos o Kafka localmente, não precisamos definir todos esses valores, pois a maioria deles já estão definidos com os valores padrões nos arquivos de configuração do Kafka, porém, para utilizar essa imagem Docker, essas variáveis de ambiente são obrigatórias.

Configurando nossas aplicações

Depois podemos definir os nossos serviços no arquivo. Eles possuem a mesma estrutura no arquivo que os serviços anteriores: primeiro, definimos o nome dos serviços, no caso `shop-api` , `shop-validator` , `shop-report` e `shop-python` . Para todos, temos que definir a imagem do docker que será utilizada, que são as que definimos na seção anterior quando criamos as imagens para as aplicações. Também precisamos definir um nome para o contêiner. Para esses três primeiros campos, eu defini sempre o mesmo nome, o que não é obrigatório, mas facilita a organização do arquivo.

Depois temos que definir as portas para os serviços e a última configuração, que é o `depends_on` , para definir que os serviços só devem ser iniciados depois que o Kafka já estiver pronto e

funcionando. Isso é importante sempre que existir dependência entre os serviços, pois o `docker-compose` iniciará todos os serviços ao mesmo tempo e, se um depender de outro, podem acontecer erros na inicialização dos serviços. O código a seguir mostra a definição dos serviços.

```
  shop-api:
    image: shop-api
    container_name: shop-api
    ports:
      - 8080:8080
    depends_on:
      - kafka

  shop-validator:
    image: shop-validator
    container_name: shop-validator
    ports:
      - 8081:8081
    depends_on:
      - kafka

  shop-report:
    image: shop-report
    container_name: shop-report
    ports:
      - 8082:8082
    depends_on:
      - kafka

  shop-retry:
    image: shop-report
    container_name: shop-report
    ports:
      - 8083:8083
    depends_on:
      - kafka

  shop-python:
```

```
    image: shop-python
    container_name: shop-python
    ports:
      - 5000:5000
    depends_on:
      - kafka
```

Executando o docker-compose

Agora basta executar o comando `docker-compose up` para que todos os contêineres sejam iniciados. A aplicação deve subir normalmente, e deverão aparecer na tela os logs de todas as aplicações juntas no console. Podemos testar a aplicação fazendo uma chamada para criar uma compra na `shop-api`. Para isso, podemos fazer uma chamada POST para o endereço `http://localhost:8080/shop` enviando o seguinte JSON:

```
{
    "buyerIdentifier": "b-3",
    "items": [
        {
            "productIdentifier": "123456789",
            "amount": "1",
            "price": "1000"
        }
    ]

}
```

Para verificar se tudo funcionou, podemos fazer uma chamada GET também para o endereço `http://localhost:8080/shop`. Veremos que a resposta dessa chamada será:

```
[
    {
        "identifier": "d1b78960-c829-4a2e-a92c-8e3ae250f0f6",
        "dateShop": "2021-10-28",
```

```
        "status": "SUCCESS",
        "buyerIdentifier": "b-3",
        "items": [
            {
                "productIdentifier": "123456789",
                "amount": 1,
                "price": 1000.0
            }
        ]
    }
]
```

Como o estado da compra está como `SUCCESS`, isso indica que o processamento foi executado com sucesso, então todas as aplicações e o Kafka estão funcionando corretamente usando o Docker.

Rodar a aplicação com o Docker não é obrigatório, mas fica muito mais fácil rodar a aplicação inteira, por isso eu recomendo tentar executar as aplicações utilizando essa ferramenta.

Agora já vimos todos os principais conceitos do Kafka sobre como fazer a comunicação de aplicações utilizando o Kafka, mas ainda existem diversos outros conceitos importantes que podemos explorar. No próximo capítulo, veremos como implementar os testes de unidade para classes que utilizam o Kafka.

Capítulo 15
Testes de unidade

Neste capítulo, veremos como criar os testes de unidade para os métodos que utilizam as classes do Kafka. Criar testes é bastante importante para garantir a qualidade da aplicação e para evitar que mudanças na aplicação façam com que códigos já existentes parem de funcionar. O ideal nos testes de unidade é que eles não dependam de nenhuma ferramenta exterior ao código, como banco de dados ou o Kafka. Por isso, é importante saber como criar testes utilizando `mocks` das classes do Kafka, para não termos que inicializar o Kafka quando formos executar os testes. Para exemplificar isso, vamos desenvolver testes de unidade em todas as aplicações desenvolvidas até aqui.

15.1 Configurando as aplicações

Para possibilitar a execução dos testes em nossas aplicações, basta adicionar a dependência `spring-boot-starter-test` com o escopo de `test` ao arquivo `pom.xml` de todas as aplicações. Essa dependência já adiciona diversas bibliotecas que serão necessárias para a criação dos testes, como o **JUnit** e o **Mockito**. O escopo define que essa biblioteca será utilizada apenas para os testes, então ela não será adicionada na versão final da aplicação.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
```

15.2 Testes na shop-api

Vamos começar adicionando os testes na `shop-api`. Temos duas classes que recebem ou enviam dados para o Kafka nesta aplicação. Vamos começar com a `SendKafkaMessage`, que é a classe que envia mensagens para o Kafka.

Ela possui apenas um método que recebe um `shopDTO` e o envia para um tópico no Kafka. Para testar esse método, vamos chamá-lo e depois confirmar que o método para o envio da mensagem do Kafka foi chamado. O código a seguir mostra o código da classe de teste criada.

```
package com.santana.events;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.Mockito;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.test.context.junit.jupiter.SpringExtension;

import com.santana.dto.ShopDTO;

@ExtendWith(SpringExtension.class)
public class SendKafkaMessageTests {

    @InjectMocks
    private SendKafkaMessage sendKafkaMessage;

    @Mock
    private KafkaTemplate<String, ShopDTO> kafkaTemplate;

    private static final String SHOP_TOPIC_NAME = "SHOP_TOPIC";

    @Test
    public void testSendMessage() {
```

```
        ShopDTO shopDTO = new ShopDTO();
        shopDTO.setStatus("SUCCESS");
        shopDTO.setBuyerIdentifier("b-1");

        sendKafkaMessage.sendMessage(shopDTO);

        Mockito
        .verify(kafkaTemplate, Mockito.times(1))
        .send(SHOP_TOPIC_NAME, "b-1", shopDTO);
    }
}
```

Note que a classe foi anotada com a `@ExtendWith(SpringExtension.class)`, indicando que essa classe fará o teste de uma aplicação do Spring Boot. A classe de teste foi chamada de `SendKafkaMessageTests`, que é o mesmo nome da classe que está sendo testada, mais o sufixo `Tests`. Isso não é obrigatório, mas é um padrão bastante seguido, para facilitar a associação das classes originais com as classes de teste.

A anotação `@InjectMocks` define o objeto da classe que será testada, no nosso caso a `SendKafkaMessage`. A `@Mock` deve ser utilizada para todas as classes que são utilizadas dentro da classe que será testada. O teste foi implementado no método `testSendMessage`.

Veja que basicamente criamos um objeto do tipo `ShopDTO`, e com ele chamamos o método que queremos testar, no caso o `sendMessage`. Por fim, verificamos se o método `send` da classe `KafkaTemplate` foi chamado exatamente uma vez. Outra anotação importante é a `@Test`, que indica que esse método é um teste de unidade do JUnit.

A outra classe que precisamos testar é a `ReceiveKafkaMessage`, por isso criamos a classe de teste `ReceiveKafkaMessageTests`. A configuração geral dessa classe é bem parecida com a anterior: temos a anotação `@ExtendWith(SpringExtension.class)` e adicionamos a anotação `@InjectMocks` em um objeto do tipo `ReceiveKafkaMessage`, que é o que será testado. Também precisamos adicionar um `@Mock`

para a classe `ShopRepository`, que é chamada na classe que será testada.

```
package com.santana.events;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.Mockito;
import org.springframework.test.context.junit.jupiter.SpringExtension;

import com.santana.dto.ShopDTO;
import com.santana.dto.ShopItemDTO;
import com.santana.model.Shop;
import com.santana.repository.ShopRepository;

@ExtendWith(SpringExtension.class)
public class ReceiveKafkaMessageTests {

    @InjectMocks
    private ReceiveKafkaMessage receiveKafkaMessage;

    @Mock
    private ShopRepository shopRepository;

    @Test
    public void testSuccessfulMessageReceived() {

        ShopDTO shopDTO = new ShopDTO();
        shopDTO.setStatus("SUCCESS");

        ShopItemDTO shopItemDTO = new ShopItemDTO();
        shopItemDTO.setAmount(1000);
        shopItemDTO.setProductIdentifier("product-1");
        shopItemDTO.setPrice((float) 100);

        shopDTO.getItems().add(shopItemDTO);
```

```
        Shop shop = Shop.convert(shopDTO);

        Mockito

.when(shopRepository.findByIdentifier(shopDTO.getIdentifier()))
            .thenReturn(shop);

        receiveKafkaMessage.listenShopEvents(shopDTO);
        Mockito
            .verify(shopRepository, Mockito.times(1))
            .findByIdentifier(shopDTO.getIdentifier());

        Mockito
            .verify(shopRepository, Mockito.times(1))
            .save(shop);
    }
}
```

O método que estamos testando agora é o `listenShopEvents`, que é o método que implementa o consumidor do tópico do Kafka. Algumas partes interessantes desse teste são a linha na qual definimos o que fazer quando for chamado o método `findByIdentifier` do objeto `shopRepository` e também as verificações que fazemos para garantir que o método funcionou corretamente, que é verificar que os métodos `findByIdentifier` e `save` do objeto `shopRepository` foram executados exatamente uma vez.

15.3 Testes na shop-validator

Vamos implementar os testes na `shop-validator` agora. Essa aplicação possui um método um pouco mais complexo, o `listenShopTopic`, que diz se uma compra é válida ou não, então precisaremos criar mais de um teste para esse método, um para a

compra que é realizada com sucesso e outro para a compra que é cancelada por ter algum erro.

O método que faz a validação da compra está implementado na classe `ReceiveKafkaMessage`, por isso vamos criar a classe de teste `ReceiveKafkaMessageTests`. A configuração dela também é bem parecida com as classes da aplicação `shop-api`. Antes dos testes, eu criei os métodos `getShopDTO` e `getProduct`, que apenas criam os objetos `ShopDTO` e `Product`. Isso foi feito porque teremos mais de um teste que usará esses objetos.

O primeiro teste está no método `testProcessShopSuccess`. Note que ele cria uma compra e um produto. Depois, utilizando o mock da classe `ProductRepository`, ele configura a resposta que o método `findByIdentifier` deve retornar. Em seguida, apenas fazemos a chamada ao método `listenShopTopic`, que é o método que será testado, e, por fim, validamos se as respostas estão corretas. Para isso, verificamos que o método de enviar uma mensagem para o Kafka foi chamado exatamente uma vez, e que o status da compra está como `SUCCESS`.

```
package com.santana.events;

import org.assertj.core.api.Assertions;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.Mockito;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.test.context.junit.jupiter.SpringExtension;

import com.santana.dto.ShopDTO;
import com.santana.dto.ShopItemDTO;
import com.santana.model.Product;
import com.santana.repository.ProductRepository;

@ExtendWith(SpringExtension.class)
```

```
public class ReceiveKafkaMessageTests {

    @InjectMocks
    private ReceiveKafkaMessage receiveKafkaMessage;

    @Mock
    private KafkaTemplate<String, ShopDTO> kafkaTemplate;

    @Mock
    private ProductRepository productRepository;

    private static final String SHOP_TOPIC_EVENT_NAME =
"SHOP_TOPIC_EVENT";

    public ShopDTO getShopDTO() {
        ShopDTO shopDTO = new ShopDTO();
        shopDTO.setBuyerIdentifier("b-1");

        ShopItemDTO shopItemDTO = new ShopItemDTO();
        shopItemDTO.setAmount(1000);
        shopItemDTO.setProductIdentifier("product-1");
        shopItemDTO.setPrice((float) 100);

        shopDTO.getItems().add(shopItemDTO);
        return shopDTO;
    }

    public Product getProduct() {
        Product product = new Product();
        product.setAmount(1000);
        product.setId(1L);
        product.setIdentifier("product-1");
        return product;
    }

    @Test
    public void testProcessShopSuccess() {

        ShopDTO shopDTO = getShopDTO();
```

```
        Product product = getProduct();

        Mockito
            .when(productRepository.findByIdentifier("product-1"))
            .thenReturn(product);

        receiveKafkaMessage.listenShopTopic(shopDTO);

        Mockito
            .verify(kafkaTemplate, Mockito.times(1))
            .send(SHOP_TOPIC_EVENT_NAME, shopDTO);

        Assertions
            .assertThat(shopDTO.getStatus()).isEqualTo("SUCCESS");

    }
}
```

Note que eu estou testando a versão do método `listenShopTopic` do capítulo 7. No capítulo 9 eu adicionei alguns parâmetros a mais nesses métodos apenas para demonstrar a utilização das chaves, mas a versão final da aplicação ficou sem esses parâmetros.

Podemos criar um outro teste que vai mostrar se o comportamento da aplicação está correto para uma compra inválida. Para isso, podemos fazer os mesmos passos do método anterior, mas agora enviar uma compra com um produto inválido. Apenas temos que mudar a validação no final, para garantir que o status da compra fique com o status `ERROR` após a execução do método `listenShopTopic`.

```
@Test
public void testProcessShopError() {

    ShopDTO shopDTO = getShopDTO();

    Mockito
        .when(productRepository.findByIdentifier("product-1"))
        .thenReturn(null);
```

```
    receiveKafkaMessage.listenShopTopic(shopDTO);

    Mockito
        .verify(kafkaTemplate, Mockito.times(1))
        .send(SHOP_TOPIC_EVENT_NAME, shopDTO);

    Assertions
        .assertThat(shopDTO.getStatus()).isEqualTo("ERROR");

}
```

15.4 Testes na shop-report

Vamos criar os testes para a aplicação `shop-report`. Essa aplicação possui apenas uma classe bastante simples que consome mensagens do Kafka, por isso os testes também serão simples. Apenas temos que verificar se a mensagem é recebida e os dados são atualizados no banco de dados.

O teste foi implementado no método `testProcessShopSuccess` e basicamente ele está verificando se o método `incrementShopStatus` da classe `ReceiveKafkaMessage` foi chamado exatamente uma vez.

```
package com.santana.events;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.Mockito;
import org.springframework.test.context.junit.jupiter.SpringExtension;

import com.santana.dto.ShopDTO;
import com.santana.repository.ReportRepository;
```

```
@ExtendWith(SpringExtension.class)
public class ReceiveKafkaMessageTests {

    @InjectMocks
    private ReceiveKafkaMessage receiveKafkaMessage;

    @Mock
    private ReportRepository reportRepository;

    public ShopDTO getShopDTO() {
        ShopDTO shopDTO = new ShopDTO();
        shopDTO.setStatus("SUCCESS");
        return shopDTO;
    }


    @Test
    public void testProcessShopSuccess() {

        ShopDTO shopDTO = getShopDTO();

        receiveKafkaMessage.listenShopTopic(shopDTO);

        Mockito
            .verify(reportRepository, Mockito.times(1))
            .incrementShopStatus(shopDTO.getStatus());

    }
}
```

15.5 Testes na shop-retry

Por último, vamos criar os testes na `shop-retry`. Nela, temos o consumidor que verifica se uma compra tem ou não itens cadastrados e, se não tiver, é lançada uma mensagem de erro no tópico para reprocessar as compras. Vamos testar dois casos nessa

aplicação: primeiro, as compras que estão corretas e, depois, as compras que possuem problemas. O teste do primeiro caso é bastante simples, podemos chamar o método com uma compra válida e, no fim, apenas garantir que o método que insere mensagens com erros no Kafka não foi chamado. O código a seguir mostra a implementação desse teste.

```
package com.santana.events;

import java.util.ArrayList;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.Mockito;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.test.context.junit.jupiter.SpringExtension;

import com.santana.dto.ShopDTO;
import com.santana.dto.ShopItemDTO;

@ExtendWith(SpringExtension.class)
public class ReceiveKafkaMessageTests {

    @InjectMocks
    private ReceiveKafkaMessage receiveKafkaMessage;

    @Mock
    private KafkaTemplate<String, ShopDTO> kafkaTemplate;

    private static final String SHOP_TOPIC_RETRY = "SHOP_TOPIC_RETRY";

    public ShopDTO getShopDTO() {
        ShopDTO shopDTO = new ShopDTO();
        shopDTO.setBuyerIdentifier("b-1");

        ShopItemDTO shopItemDTO = new ShopItemDTO();
        shopItemDTO.setAmount(1000);
```

```
        shopItemDTO.setProductIdentifier("product-1");
        shopItemDTO.setPrice((float) 100);

        shopDTO.getItems().add(shopItemDTO);
        return shopDTO;
    }

    @Test
    public void testProcessShopSuccess() {
        ShopDTO shopDTO = getShopDTO();
        receiveKafkaMessage.listenShopTopic(shopDTO);

        Mockito
            .verify(kafkaTemplate, Mockito.never())
            .send(SHOP_TOPIC_RETRY, shopDTO);
    }

}
```

No segundo caso, quando acontece algum erro no processamento, será chamado o produtor do Kafka para adicionar uma mensagem a fim de fazer o seu reprocessamento, então basta verificar que o método de inserir mensagens no Kafka foi chamado exatamente uma vez. Note que, antes de chamar o método `listenShopTopic`, eu limpei a lista de itens da compra, exatamente para forçar o erro na execução do método.

```
@Test
public void testProcessShopError() {
    ShopDTO shopDTO = getShopDTO();
    shopDTO.setItems(new ArrayList<>());
    receiveKafkaMessage.listenShopTopic(shopDTO);

    Mockito
        .verify(kafkaTemplate, Mockito.times(1))
        .send(SHOP_TOPIC_RETRY, shopDTO);
}
```

Criar testes é bastante importante para garantir a qualidade da aplicação, por isso recomendo sempre criar testes para todas as classes que consomem ou produzem mensagens no Kafka. Como mostrado aqui, criar os mocks para as classes do Kafka é tranquilo, então não existe nenhuma dificuldade para criar os testes nessas classes.

Agora que nossas aplicações estão prontas e testadas, vamos ver uma última funcionalidade do Kafka, que é o processamento de fluxos de dados (ou streams de dados). Neste caso, os dados são recuperados de um tópico Kafka, alguns processamentos são feitos nesses dados como agregações e filtragens e, por fim, os dados são salvos novamente em outra fila do Kafka. Veremos como implementar o processamento de fluxos de dados no Kafka no próximo capítulo.

Capítulo 16
Kafka Streams

Neste livro, focamos na utilização do Kafka para a comunicação entre diferentes microsserviços, porém existe outra funcionalidade do Kafka que é bastante interessante e que vale a pena conhecer, pois ela pode resolver diversos problemas. Estamos falando do processamento de fluxos de dados (ou *streams*) que estão em um tópico. Esse tipo de processamento normalmente utiliza algumas funções para que sejam feitas transformações nos dados como agrupamentos, agregações e filtros de uma forma fácil de implementar.

O Kafka possui ferramentas para esse tipo de processamento, e existe uma biblioteca Java que facilita bastante o desenvolvimento desse tipo de processamento. Neste capítulo, veremos como implementar o processamento de alguns fluxos nas aplicações que desenvolvemos até aqui, por exemplo, para contar quantas compras foram feitas, qual o total de vendas e quantas compras foram feitas para cada produto.

Além disso, com esse tipo de processamento, é possível salvar o resultado do processamento de diversas formas como em um banco de dados, em uma ferramenta de logs, ou até mesmo inserir o resultado em outro tópico do Kafka.

16.1 Configuração da aplicação

Nesta aplicação, precisaremos de quatro dependências: o `kafka-clients`, que já utilizamos no capítulo 11 e que serve para conectar ao Kafka; o `kafka-streams`, que faz o processamento do fluxo de dados; o `gson`, que faz a serialização e deserialização de string para

JSON; e o `lombok`, que também já utilizamos nas aplicações anteriores. O XML a seguir mostra o arquivo `pom.xml` dessa aplicação.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.santana</groupId>
    <artifactId>kafka-stream</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <dependencies>
        <dependency>
            <groupId>org.apache.kafka</groupId>
            <artifactId>kafka-streams</artifactId>
            <version>2.8.0</version>
        </dependency>
        <dependency>
            <groupId>org.apache.kafka</groupId>
            <artifactId>kafka-clients</artifactId>
            <version>2.8.0</version>
        </dependency>
        <dependency>
            <groupId>com.google.code.gson</groupId>
            <artifactId>gson</artifactId>
            <version>2.8.9</version>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.20</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
</project>
```

Também precisaremos da classe `ShopDTO` que criamos nas aplicações anteriores. Ela será necessária porque, nos fluxos, processaremos as compras que foram enviadas para o tópico `SHOP_TOPIC_EVENT`. O código a seguir mostra a implementação dessa classe.

```
package com.santana.dto;

import lombok.Getter;
import lombok.Setter;
import lombok.ToString;

@Getter
@Setter
@ToString
public class ShopDTO {
    private String identifier;
    private String status;
    private String buyerIdentifier;
}
```

A única novidade dessa classe aqui é a anotação `@ToString`, que pertence ao Lombok. Ela gera o método `toString` para a classe, incluindo todos os atributos da classe no método.

Serialização/Deserialização

Para a utilização do `kafka-streams`, não estamos utilizando o Spring Boot, então a configuração da aplicação é um pouco mais complexa. Devido a isso, para fazer a leitura das compras que estão nos tópicos do Kafka, precisaremos implementar três classes para a serialização/deserialização dos dados. Utilizaremos o `Gson` que é uma biblioteca para ler JSON e transformar em objetos de forma simples.

A primeira classe que vamos implementar é a `ShopDeserializer`. Ela transforma bytes no nosso objeto `ShopDTO`, isso é feito no método `deserialize`. Nele, podemos ver que o método recebe um array de bytes e o transformamos em um objeto `ShopDTO` chamando o

método `fromJson`. Esse método será chamado diretamente pelo `kafka-streams` quando as mensagens do Kafka são lidas.

```
package com.santana.streams.serializer;

import java.nio.charset.Charset;

import org.apache.kafka.common.serialization.Deserializer;

import com.google.gson.Gson;
import com.santana.dto.ShopDTO;

public class ShopDeserializer implements Deserializer {

    private static final Charset CHARSET =
        Charset.forName("UTF-8");
    static private Gson gson = new Gson();

    @Override
    public Object deserialize(String s, byte[] bytes) {
        try {
            String shop =
                new String(bytes, CHARSET);

            return gson
                .fromJson(shop, ShopDTO.class);
        } catch (Exception e) {
            throw new IllegalArgumentException(
                "Error reading bytes! ",
                e);
        }
    }

}
```

Também temos o método contrário, o que transforma um objeto do tipo `ShoDTO` para um array de bytes. Estes são os dados que serão

enviados para o Kafka pelo `kafka-streams`:

```
package com.santana.streams.serializer;

import java.nio.charset.Charset;

import org.apache.kafka.common.serialization.Serializer;

import com.google.gson.Gson;

public class ShopSerializer implements Serializer {

    private static final Charset CHARSET
        = Charset.forName("UTF-8");
    static private Gson gson = new Gson();

    @Override
    public byte[] serialize(String s, Object o) {
        String line = gson.toJson(o);

        return line.getBytes(CHARSET);
    }

}
```

Por fim, temos que criar uma classe do tipo `Serde` (que significa Serialização/Deserialização), que é um tipo de classe do `kafka-streams`. Ela apenas mapeia quais são as classes que serão utilizadas para fazer a Serialização/Deserialização dos objetos que estão nos tópicos do Kafka. Note que ela apenas retorna objetos das classes que criamos anteriormente.

```
package com.santana.streams.serializer;

import org.apache.kafka.common.serialization.Deserializer;
import org.apache.kafka.common.serialization.Serde;
import org.apache.kafka.common.serialization.Serializer;

public class ShopSerde implements Serde<Object> {
```

```
    private ShopSerializer shopSerializer
        = new ShopSerializer();

    private ShopDeserializer shopDeserializer
        = new ShopDeserializer();

    @Override
    public void close() {
        shopSerializer.close();
        shopDeserializer.close();

    }

    @Override
    public Serializer serializer() {
        return shopSerializer;
    }

    @Override
    public Deserializer deserializer() {
        return shopDeserializer;
    }
}
```

16.2 Processamento de fluxos

Agora que a configuração da aplicação e as classes para fazer a serialização/deserialização dos dados estão prontas, podemos implementar os processamentos que faremos com o `kafka-streams`. Normalmente, esse tipo de aplicação é desenvolvido em um método `main` que fica conectado em um tópico do Kafka esperando que as mensagens cheguem para fazer o processamento.

Veremos que nessas aplicações poderemos fazer diversos tipos de operações, salvando os dados na própria aplicação, e depois

podemos visualizar os resultados de diversas formas, por exemplo, apenas imprimindo no console da aplicação ou enviando os resultados para outro tópico do Kafka. Vamos então implementar alguns fluxos.

Imprimindo compras

Vamos implementar um primeiro fluxo bem simples, ele apenas imprimirá as compras que foram feitas no console. Inicialmente, precisamos adicionar quatro propriedades na nossa aplicação que são:

- `APPLICATION_ID_CONFIG` : Um identificador da aplicação, que pode ter qualquer nome. Ele é importante também, pois será criado um `consume group` com o nome da aplicação.
- `BOOTSTRAP_SERVERS_CONFIG` : O endereço do Kafka ao qual a aplicação vai se conectar; no nosso caso, `localhost:9092` .
- `DEFAULT_KEY_SERDE_CLASS_CONFIG` : A classe que vai fazer a serialização/deserialização da chave das mensagens. Aqui vamos utilizar a classe do próprio `kafka-streams` , que é a `Serdes.String().getClass()` .
- `DEFAULT_VALUE_SERDE_CLASS_CONFIG` : A classe que vai fazer a serialização/deserialização do valor das mensagens. Aqui usaremos a classe que desenvolvemos para ler as compras que são enviadas para o tópico, a `ShopSerde` .

Depois da configuração, que será igual para todas as aplicações, iniciamos a declaração do processamento que será feito. Duas linhas serão sempre obrigatórias: a primeira, que cria o objeto que inicia a declaração, da classe `StreamsBuilder` , e a segunda, que indica o tópico que será lido, a `builder.stream(SHOP_TOPIC_EVENT)` . Após isso, iniciamos a declaração das operações que serão feitas nos dados. Nessa primeira aplicação vamos apenas imprimir no console todas as compras que chegarem no tópico, com a chamada `inputTopic.print(Printed.toSysOut())` .

Uma nota importante: até aqui nenhum processamento foi efetivamente realizado. No `kafka-streams`, primeiro declaramos todas as operações que queremos fazer, e só depois iniciamos realmente o processamento.

Por fim, criamos o objeto `streams` da classe `KafkaStreams` e chamamos o método `start`. Depois que essa linha for executada, o processamento será iniciado.

```
package com.santana.streams;

import java.util.Properties;

import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.KafkaStreams;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.kstream.KStream;
import org.apache.kafka.streams.kstream.Printed;

import com.santana.dto.ShopDTO;
import com.santana.streams.serializer.ShopSerde;

public class PrintShops {

    private static final String SHOP_TOPIC = "SHOP_TOPIC";

    public static void main(String [] args) {
           Properties props = new Properties();
           props.put(
                StreamsConfig.APPLICATION_ID_CONFIG,
                "show-shops");
           props.put(
                StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
                "localhost:9092");
           props.put(
                StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
                Serdes.String().getClass());
           props.put(
```

```
                StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
                ShopSerde.class.getName());

        StreamsBuilder builder = new StreamsBuilder();
        KStream<String, ShopDTO> inputTopic =
                builder.stream(SHOP_TOPIC);

        inputTopic
                .print(Printed.toSysOut());

        KafkaStreams streams =
            new KafkaStreams(builder.build(), props);
        streams.start();
    }
}
```

A saída desse programa será a impressão das compras que são realizas em nossa aplicação, como no exemplo a seguir:

```
[KSTREAM-SOURCE-0000000000]:
    b-3, 82120fc5-fa1d-418a-a225-3fda61e86316
[KSTREAM-SOURCE-0000000000]:
    b-3, 307fc4c2-986c-4261-92df-6fef3d90b2cc
[KSTREAM-SOURCE-0000000000]:
    b-3, 5b8ef62f-e971-4d18-8ec0-1f070a74ce21
```

Compras por usuários

Outro processamento que podemos criar é contabilizar o número de compras por comprador, lembrando que temos o identificador do comprador como chave da mensagem. Para isso, temos que utilizar o método `groupByKey()`, que agrupa as mensagens pela chave e, depois, o método `count` para contar o número das mensagens que foram agrupadas com uma chave.

Passamos como parâmetro para o método `count` um objeto do tipo `Materialized`, que indica como os dados serão armazenados na memória durante a execução da aplicação. Se não passarmos esse parâmetro, serão contadas apenas as mensagens que forem

processadas juntas, e não serão acumuladas todas as mensagens que serão processadas durante a aplicação.

```
public static void main(String [] args) {

    Properties props = new Properties();
    props.put(
        StreamsConfig.APPLICATION_ID_CONFIG,
        "count-shops-by-users");
    props.put(
        StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
        "localhost:9092");
    props.put(
        StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
        Serdes.String().getClass());
    props.put(
        StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
        ShopSerde.class.getName());


    StreamsBuilder builder = new StreamsBuilder();
    KStream<String, ShopDTO> inputTopic =
           builder.stream(SHOP_TOPIC);

    KTable<String, Long>  comprasPorUsuario = inputTopic
           .groupByKey()
           .count(Materialized.as("count-store"));

    comprasPorUsuario
        .toStream()
        .print(Printed.toSysOut());

    KafkaStreams streams =
        new KafkaStreams(builder.build(), props);
    streams.start();

}
```

Sempre que recebermos algumas mensagens no tópico, o Kafka fará a agregação, contando o número de compras feitas por cada usuário, então sempre que enviarmos algumas compras será impresso o número de compras.

Neste exemplo, quando iniciamos a aplicação, o usuário "b-3" já tinha feito 16 compras, depois ele fez mais 5 e depois mais uma, finalizando com 22 compras. Já o usuário "b-1" não tinha feito nenhuma compra, mas depois de algum tempo ele fez 4 compras, e depois mais 7, totalizando 11 compras durante a execução da aplicação.

Note que o Kafka não processa mensagem por mensagem sempre. Se durante um intervalo de tempo forem feitas várias compras, o Kafka processará todas de uma vez.

```
[KTABLE-TOSTREAM-0000000002]: b-3, 16
[KTABLE-TOSTREAM-0000000002]: b-3, 21
[KTABLE-TOSTREAM-0000000002]: b-3, 22
[KTABLE-TOSTREAM-0000000002]: b-1, 4
[KTABLE-TOSTREAM-0000000002]: b-1, 11
```

16.3 Enviando resultados para outros tópicos

Nos exemplos anteriores, apenas imprimimos os resultados no console da aplicação, mas outra funcionalidade bastante interessante do `kafka-streams` é que podemos enviar os resultados do processamento para outro tópico. Para imprimir os resultados no console, estávamos usando o trecho de código a seguir:

```
comprasPorUsuario
    .toStream()
    .print(Printed.toSysOut());
```

Alterar isso para que os resultados sejam enviados para um tópico do Kafka é bastante simples. Basta substituir as linhas anteriores

por:

```
stream
     .toStream()
    .to("CountUsersByKey",
        Produced.with(Serdes.String(),
        Serdes.Long()));
```

Assim os dados serão enviados para o tópico `stream-teste`.

16.4 Janelas de tempo

A última funcionalidade do `kafka-streams` que veremos é a de dividir os dados em janelas de tempo. Podemos pegar todas as compras dos usuários divididos em intervalos de 5 segundos, por exemplo e isso pode ser utilizado em diversas funcionalidades. Para verificar se existe duplicação nos registros, poderíamos verificar se em um intervalo tivemos registro iguais de um mesmo usuário.

Para usar essa funcionalidade, temos apenas que adicionar ao processamento a chamada ao método `windowedBy()` passando um intervalo de tempo. No exemplo a seguir, estamos usando cinco segundos.

```
KTable<Windowed<String>, Long>  stream = inputTopic
    .groupByKey()
    .windowedBy(TimeWindows.of(Duration.ofSeconds(5)))
    .count();
```

Com isso, o resultado do processamento será parecido, mas note que agora, além de agrupar pelo usuário que fez a compra, a chave da mensagem também possui dois números. Como exemplo, temos `1639963360000/1639963365000`, que formam o intervalo de tempo em que as mensagens foram processadas.

```
[KTABLE-TOSTREAM-0000000003]:
    [b-1@1639963360000/1639963365000], 1
[KTABLE-TOSTREAM-0000000003]:
    [b-1@1639963365000/1639963370000], 6
[KTABLE-TOSTREAM-0000000003]:
    [b-1@1639963370000/1639963375000], 4
[KTABLE-TOSTREAM-0000000003]:
    [b-1@1639963375000/1639963380000], 1
```

16.5 Conclusão

Espero que você tenha gostado do que viu até aqui, que tenha aprendido e entendido bem os principais conceitos do Kafka e que consiga aplicar o que foi mostrado aqui no seu dia a dia. Como falado no começo do livro, o processamento assíncrono não substitui o síncrono, mas ele pode facilitar muito o desenvolvimento de algumas funcionalidades e tornar um sistema computacional muito mais robusto e confiável. Sem dúvida é muito importante que as pessoas programadoras conheçam bem esse tipo de processamento e conheçam algumas ferramentas que o possibilitem, como o Kafka.